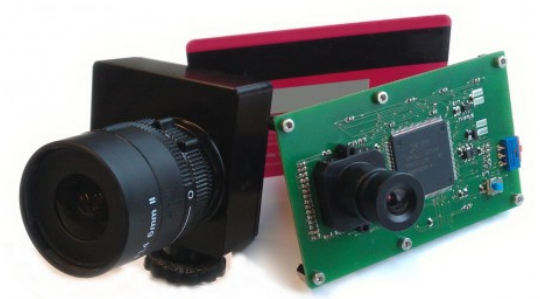
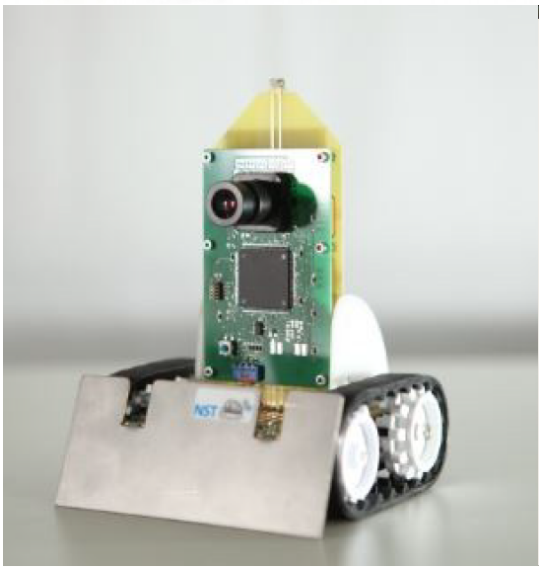
# VISUAL INFORMATION PROCESSING PROJECTSEMINAR

ANDREAS ROTTACH

Optic-flow-based robot navigation

Wintersemester 2016/17





University Ulm

# CONTENTS

# ACRONYMS

DVS Dynamic Vision Sensor

EDVS Embedded Dynamic Vision Sensor

PUSHBOT small robot platform

CUDA Compute Unified Device Architecture

# 1 INTRODUCTION

The two goals when developing this framework are the implementation of a real-time optic-flow estimation with a Dynamic Vision Sensor (DVS) camera as described in [1] and afterwards the biologically inspired navigation of a small robot platform (PushBot) similar to the navigation principals of bees. Their visual system and navigation behavior is described in detail in [2].

## 1.1 EVENT-BASED CAMERAS

In contrast to conventional cameras, which capture their environment frame by frame, a DVS is event-based and works similar to the human retina. Instead of producing frame-based data, which waists memory and energy, the DVS transmits pixelwise intensity changes. This result in no data transfer when capturing a static scene. Only movements and changes in brightness in the camera's visual field of view generate events. Therefore and because of the much lower bandwidth, the DVS is able to operate much faster compared to conventional frame-based camera.

## 1.2 OPTIC-FLOW ESTIMATION WITH EVENT-BASED CAMERAS

In this work, the findings from [1] are taken to design a optic-flow estimation system. The estimation step depends in general on spatial-temporal filters which are able to detect motions with a specific speed and orientation.

The spatial component of such a filter is defined by the real respectively imaginary component of a gabor filter as shown in Equation 1 to 3 and Figure 1b, 1a. These two filter components $G_{odd}$ and $G_{even}$ are then combined with a mono- or bi-phasic temporal function, defined in Equation 4 and 5 and visualized in Figure 1c. This combination step then creates four different spatial-temporal filters (Figure 1d to 1g) which are not yet sensitive for a specific motion speed.

$$\begin{aligned} G_{\sigma,f_x^0,f_y^0}(x,y) &= \frac{2\pi}{\sigma^2}\cdot\exp\left[2\pi j(f_x^0 x + f_y^0 y)\right]\cdot \\ &\quad \exp\left[-\frac{2\pi^2\cdot(x^2+y^2)}{\sigma^2}\right] && (1) \\ G_{even} &= \Re(G_{\sigma,f_x^0,f_y^0}) && (2) \\ G_{odd} &= \Im(G_{\sigma,f_x^0,f_y^0}) && (3) \end{aligned}$$

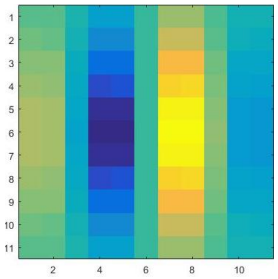
$$T_{mono}(t) = G_{\sigma_{mono},\mu_{mono}}(t) \tag{4}$$

$$T_{bi}(t) = -s_1 \cdot G_{\sigma_{bi1},\mu_{bi1}}(t) + s_2 \cdot G_{\sigma_{bi2},\mu_{bi2}}(t) \tag{5}$$

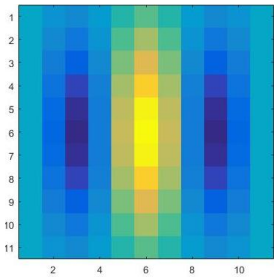
$$G_{\sigma,\mu} = \exp\left(-\frac{(t-\mu)^2}{2\sigma^2}\right) \tag{6}$$

By adding or subtracting two of this spatial-temporal filters, the resulting filter is sensitive for a specific motion speed and orientation as shown in Figure 1h, but simply taking one of these filters to detect a motion is not very suitable, because these filters apart from each other are phase dependent.

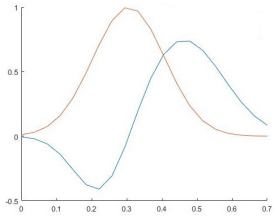
A moving edge of pixels produces a positive response when the contributing events are located in a filter region with positive coefficients (displayed as yellow values). But if the edge is shifted in the spatial dimension is such a way, that the edge is located in a filter region with negative coefficients (blue values) then the filter produces a negative output for the same moving edge. It would be desirable that the sign of the filter response is equal for all moving edges with the same speed and orientation.



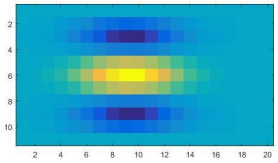
(a) $G_{odd}$



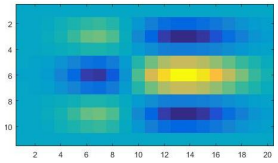
(b) $G_{even}$



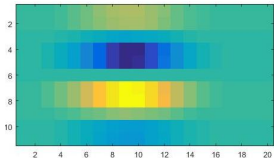
(c) $T_{mono}$: red and $T_{bi}$: blue



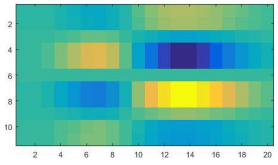
(d) $G_{even}$ and $T_{mono}$



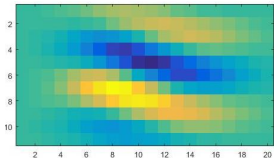
(e) $G_{even}$ and $T_{bi}$



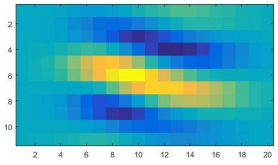
(f) $G_{odd}$ and $T_{mono}$



(g) $G_{odd}$ and $T_{bi}$



(h) $F_1(x,y,t)$ (see Equation 7).



(i) $F_2(x,y,t)$ (see Equation 8).

Figure 1: Example filter construction with a resolution of $11 \times 11 \times 20$(X × Y × T). Filter parameters are: $\sigma = 20, f_0 = 0.15, \mu_{bi1} = 0.23$. All other parameters are taken from [1] without modification. All three-dimensional filters (1d to 1i) are displayed as XT-slice with Y = 5.

This is achieved by computing the motion energy as described in [3]. By taking a pair of two speed sensitive spatial-temporal filters (Equation 7 and 8), whose filter coefficients are shifted approx. 90 degrees out of phase (called quadrature pair), both filters respond to the same moving edge with an similar intensity but opposite sign (Figure 1h and 1i). A phase independent response (motion energy) can then be extracted by squaring and summing the two filter responses $F_{1,out}$ and $F_{2,out}$ (Equation 9) after a convolution with $F_1$ and $F_2$ and the event data.

$$F_1(x,y,t) = G_{odd}(x,y) \cdot T_{bi}(t) + G_{even} \cdot T_{mono}(t) \quad (7)$$

$$F_2(x,y,t) = G_{odd}(x,y) \cdot T_{mono}(t) - G_{even} \cdot T_{bi}(t) \quad (8)$$

$$E(x',y',t') = F_{1,out}(x',y',t')^2 + F_{2,out}(x',y',t')^2 \quad (9)$$
$$\forall x', y' \in [0, 127], \forall t' \in [0, \inf)$$

The motion energy $E(x',y',t')$ at the image location $(x',y')$ and time $t'$ depends on the contrast of a moving edge: The more events contribute to the same edge, the more events contribute to the convolution and the filter response increases. To eliminate this effect, the motion energy has to be normalized according to [1]. The exact normalization step is explained in detail in the paper and implemented in this work with the parameter specified in [1]. After applying the normalization, the motion energy for each speed and orientation is scaled to approximately 1 and therefore the response is contrast independent.

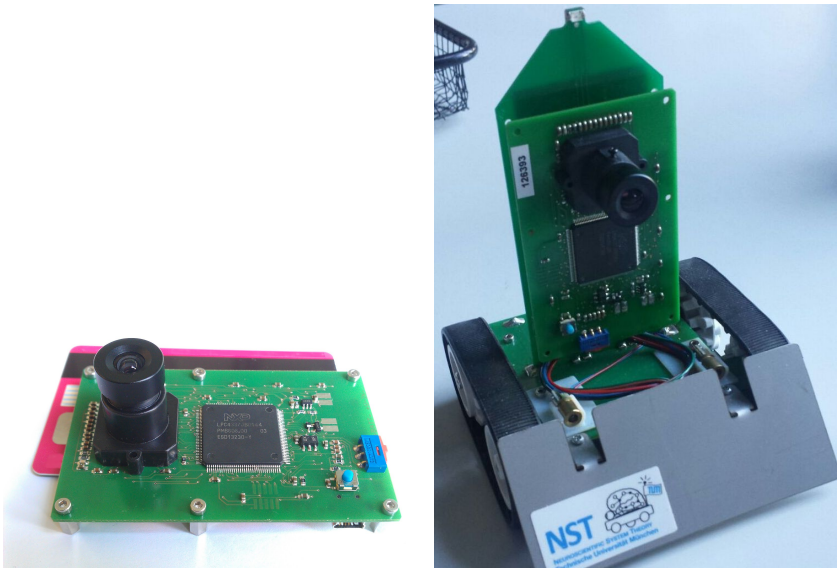
In this work, the spatial-temporal filters used, to compute the optic-flow are quite big. The filters have a spatial resolution of 11 x 11 and a temporal resolution of 20 as already shown in Figure 1. Spatially smaller convolution kernels would work as well, but with a drastic loss in quality. Shrinking the filter along its temporal axis increases the event integration in each temporal filter-slice. This decreases the ability to distinguish between similar motion speeds.

By applying the fourier transform to these filters as described in [1], it is possible to calculate their preferred speed: for the filter, shown in Figure 1 with the parameters $\sigma = 20, f_0 = 0.15, \mu_{bi1} = 0.23$ and all other parameters taken from [1], the fourier transform results in a preferred speed of $5.5 \frac{px}{\text{temporal filter size}}$. By scaling the filters temporal axis to a time window of 50 to 400 milliseconds, it is possible to detect a big variety of different speeds from 13.75 up to 110 $\frac{px}{second}$ without changing the filter shape too much.

## 1.3 USED HARDWARE

In this work the DVS is mounted on an Embedded Dynamic Vision Sensor (eDVS)-Chip that also controls the PushBot (Figure 2). The system contains a PID-controlled motor on the left and right to navi-

gate through its environment and can either control it self by executing the computations on the provided ARM 32-bit dual core CPU (by replacing the firmware) or the existing firmware can be used to control the PushBot over UART and a serial port, emulated by USB. It is also possible to keep a wireless connection between the PushBot and a computer by using a WIFI module but due to many issues and obstacles with this piece of hardware, the final implementation of this framework uses a simple USB cable to communicate with the PushBot. Additionally, the included 32-bit CPU is not powerful enough to do the necessary computations in real-time.




(a) eDVS-Chip detached from PushBot.

(b) eDVS-Chip attached to PushBot.

Figure 2: PushBot and eDVS chip used in this project.

Therefore, the necessary computations have to be done on an external computer. The system is written in C++ running on Linux and uses the Qt 5.8 framework to provide a graphical user interface. Qt is additionally used to simplify fundamental components such as inter-thread communication and the serial port handling. And because a CPU is still too slow to do all the processing, the heavy computations are done on an Nvidia GPU with Compute Unified Device Architecture (CUDA) 8.0.

The system was tested on a middle-class laptop with a Nvidia GForce GTX 840M mobile graphics card and an Intel Core i5 CPU. Due to the limited power and therefore a significant loss of event data, because of too slow processing, the system would perform much better on a high-end workstation.

# 2 PROCESSING FRAMEWORK

## 2.1 FRAMEWORK OVERVIEW

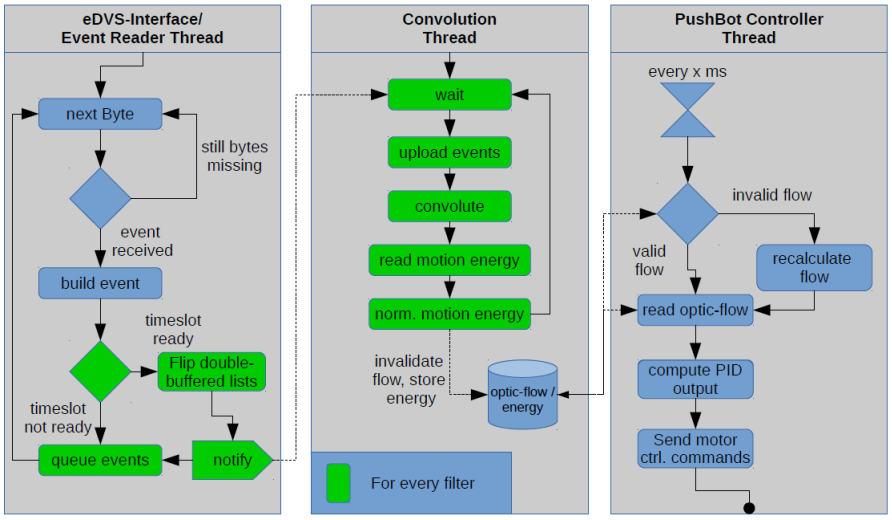



Figure 3: The proposed framework separated into three concurrent threads: Event Reader Thread, Convolution Thread and PushBot Controller Thread. The parts labeled in green run basically in parallel for all different spatial temporal filters.

To illustrate the involved steps, the view on the system is grouped into three concurrent threads which handle all incoming data as shown in Figure 3 (The fourth UI thread is omitted in this system view, because it does not contribute to the general characteristics of the system). The three different components are:

- The preparation of incoming data: Event Reader Thread
- The actual processing: Convolution Thread
- The system output: PushBot Controller Thread

For completeness, a class diagram with notes is added in the appendix in Chapter 5. It visualizes the actual system layout from a different point of view.

## 2.2 DVS EVENT DATA PARSING

The event reader thread, shown in Figure 3 on the left, is responsible for processing the incoming data, produced by the PushBot. The

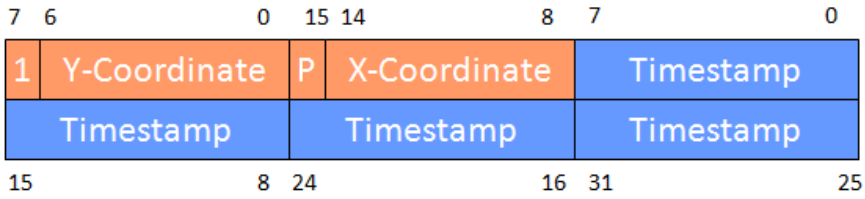



Figure 4: Event format for eDVS events: The data is stored on the PushBot and transmitted as little endian (byte with the least significant bit is transmitted first). Bytes are displayed in the order of arrival and their bit position is displayed with small numbers. Each event is transmitted as a single 16-bit (orange) and 32-bit value (blue). The 16-bit value contains a leading 1-bit, 7-bit Y coordinate, polarity bit, 7-bit X coordinate. The 32-bit timestamp is transmitted with a resolution of 1 microsecond.

data is transmitted over a serial connection as byte stream and has to be interpreted according to a predefined data format. The PushBot provides 5 different streaming formats but due to an inconsistent documentation the current implementation uses the most inefficient format which transmits 6 bytes of data per event as shown in Figure 4: 2 bytes for a 1-bit polarity and the 7-bit x and y coordinates (0 – 127) and 4 bytes for the timestamp with 1 microseconds resolution.

There is a data format that allows the system to only send the difference between subsequent timestamps and the data per event would shrink down to 3-6 bytes but again, due to the limited time and the imprecise documentation, it was not possible to get the system running with anything other then the 6 byte format even if all different format types are implemented in the system.

The system now groups the parsed events into so-called time slots. These time slots can be extracted from the different spatial-temporal filters. These filters are stored in a discrete grid and therefore their resolution is limited. The temporal resolution and the time window for a filter define its time slot size. For example, the filter, displayed in Figure 1, with a temporal resolution of 20 coefficients and a time window of 100 milliseconds integrates all events whose timestamps are less then 5 ms apart from each other. These events happen basically at the same time for this particular spatial-temporal filter and hence the time slot size for this filter is 5 ms. The system can group all events in this time slot together.

This technique has another positive effect: Instead of sending each event individually to the convolution thread which would block the threads several thousand times a second (due to synchronization), the system is able use the event grouping to minimize the synchronization overhead necessary to transfer the data between the two threads. Since filters with different preferred speeds very likely have different time windows, the grouping procedure has to happen in parallel for

different filters. On the other hand, filters were only the orientation changes (now called filter set) can still share the same event group as the time window does not change when changing the filter orientation.

The grouping is implemented as displayed in Figure 5. The system uses a double buffered list to optimize the grouping even more. It would be very inefficient to copy the whole list of events for each time slot and filter set to the convolution thread. Instead, the systems keeps two linked lists (read/write lists) of which one list is used by the event reader thread to store the data for the next time slot (write list). The second list is used by the convolution thread to read the events for processing (read list) as shown in Figure 5a. The two threads are only blocked for a short period of time, when the next time slot of events is fully parsed and the write list is filled (Figure 5b). Then, the pointers to these lists are swapped by the event reader thread and the convolution thread gets notified to start the convolution on the new event list (Figure 5c).

This allows a very fast parsing and processing of events and minimizes copy operations.

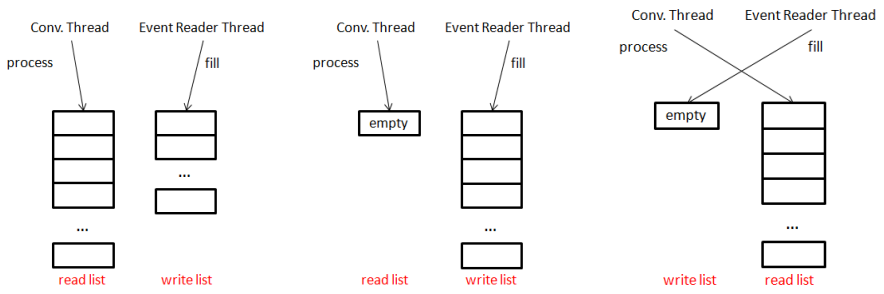



Figure 5: This figure illustrates the general procedure involved into grouping the events. Initially both threads work on their own. The convolution thread processes the event data and the event reader thread fills a new list with events (5a). Eventually, the event reader thread finishes a new time slot (5b), swaps the list pointers and everything starts from the beginning (5c).

## 2.3 CUDA-BASED OPTIC-FLOW ESTIMATION

The system is now able to read incoming events from a data stream and it groups them into logical chunks of data as explained in Section 2.2. Afterwards, the events are passed to the convolution thread in a very efficient way. The next step is to detect the motion present in the scene by processing spatial-temporal filters as described in Section 1.2. This is the most critical and computation expensive part in

this work and hence it is moved to a separate thread, shown in the center of Figure 3.

A short example demonstrates the problem: Assume a setup were the system tries to detect three different speeds with four orientations each. This results in 24 convolutions per event (two convolutions per speed and orientation). The time slot size in this setup could be something between 5 to 15 ms with up to 200 respectively 700 events per time slot. This results in up to 24*700 = 16800 convolutions in 15 ms and a total of 1.120.000 convolutions per second.

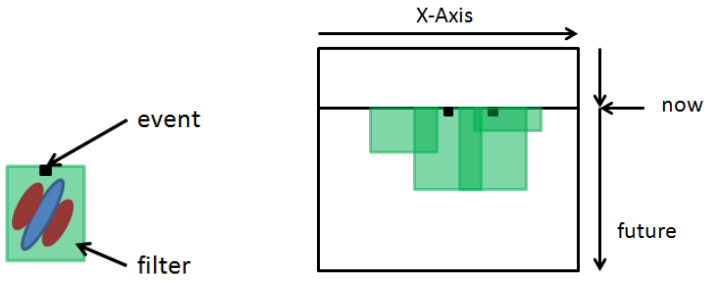
From this it follows that optimizing the filter processing was the most important step in developing this framework. The first step of a sequence of optimizations is the convolution itself.

### 2.3.1 *Forward looking ring buffer*

Usually a 2D convolution with dense image data is done by sliding the convolution kernel over the whole image and computing the filter output at each location. With 3D convolution kernels and image sequences, this problem gets even worse because the kernel has to be moved over all images to get a filter response at any spatial and temporal location. Luckily when using DVS cameras, the output is a sparse stream of events instead of full image frames and therefore sliding the convolution kernels over the full image is not necessary anymore. But we would have to slide the convolution buffer at any spatial location, where events in the past are still in the "range" of the convolution kernel. Hence the general convolution approach would still be very computational expensive.

To avoid this problem, the system uses a different approach to compute the filter response for a spatial-temporal filter. The convolution is done by using a spatial-temporal ring buffer that accumulates the filter coefficients for all incoming events as shown in Figure 6. Instead of touching an event multiple times, this approach increments the ring buffer values for each event by the corresponding filter coefficients. After all events for the current time slot (the ring buffer index) are processed, the spatial "image" at the current ring buffer index contains the filter response for the current time slot. Therefore, the response for a movement is delayed by one time slot, but compared to the slow navigation loop (described in Section 2.4) to control the PushBot, this isn't an issue.

After reading the filter response from the ring buffer, the current time slot is reset to zero and the ring buffer index is incremented at least by one (the system jumps to the next time slot with at least one event). The ring buffer size depends only on the input image size and the temporal filter resolution. In this implementation the ring buffers have a size of $128 \times 128 \times 20$.




(a) Draft of a spatial-temporal filter. (b) Proposed convolution ring buffer.

Figure 6: Draft of XT-slice of spatial-temporal filter displayed in 6a. The filter is centered spatially on the event and at $t = 0$ along the temporal axis. The spatial-temporal ring buffer (XT-slice) is displayed in 6b. The filter coefficients for each event are added to the ring buffer (displayed as half-transparent green boxes).

### 2.3.2 *Optimization 1*

This whole ring buffer architecture is directly implemented as CUDA kernel. The first implementation applied the procedure to each event separately. But due to the overhead in starting a kernel execution, this approach is not fast enough to process hundreds of events per time slot and multiple filters in parallel. Hence, the actual implementation uses the fact that events are grouped in time slots where the events timestamps are basically the same. Instead of starting the kernel for each event, it is only called once per time slot. Now each CUDA thread processes one filter coefficient for all events in the current time slot as shown in Algorithm 1.

**Algorithm 1** Batch Processing Approach

```
1  Input: Event list, Ring buffer, Ring buffer index,
       Spatial-temporal filter
2
3  Compute thread index t_idx and convert it to 3D filter
       position pos_filter.
4  Read filter value v_filter at pos_filter.
5
6  For event e in Event list
7    Compute ringbuffer pos pos_rb by using ring buffer
         index, event e and filter position pos_filter.
8    Atomic increment of ringbuffer at pos_rb by filter
         value v_filter.
9  End
```

### 2.3.3 *Optimization 2*

Algorithm 1 allows an additional optimization step. Accessing the global GPU memory is the most expensive operation when developing a CUDA kernel. It is several hundred times slower to access a value from the global memory than doing any integer or floating point operation. Furthermore, the most efficient way to access global memory is to use a coalesced access pattern where memory access is arranged in such a way, that neighboring threads access neighboring memory locations (Coalesced memory access is a bit more complex, but this is enough to describe the problem). If this is the case, the GPU can group memory operations into blocks by reading bigger chunks of data at once. This lowers the number of individual memory operations and highly increases performance. At the moment all threads read the same event data at the same time. This data is broad-casted by the GPU but it is possible to optimize this operation even more.

CUDA offers a user-managed cache, called shared memory, which can be used to optimize memory access patterns. Instead of reading the same event with all threads, it is more efficient to read different events with each thread from the global memory and store the values in the shared memory: thread one reads event one, thread two reads event two, ... and so on. All these memory operations happen in parallel and because they are coalesced, these operations are very fast. Once the data is stored in the shared memory, all threads can access the data in almost no time.

Due to the fact, that the shared memory size per block (a group of threads) is limited it is not always possible to read all events into the shared memory at once. Therefore the CUDA kernel processes the events in chunks of 128 events. An abstract summary of the whole algorithm is displayed in Algorithm 2.

This algorithm has still one downside: The atomic write operations to increment the ring buffer values depend on the event location and therefore their memory access pattern is completely unpredictable. This can't be optimized furthermore with this approach.

**Algorithm 2** Optimized Batch Processing Approach

```
1  Input: Event list, Ring buffer, Ring buffer index,
       Spatial-temporal filter
2
3  Compute thread index t_idx and convert it to 3D filter
       position pos_filter.
4  Read filter value v_filter at pos_filter.
5
6  For number of event chunks
7    Load current chunk of events in shared memory in
         parallel with as many threads as possible.
8
```

```
9     Synchronize threads and wait for memory operations
          to finish.
10
11    For event e in shared event list
12      Compute ringbuffer pos pos_rb by using ring buffer
            index, event e and filter position pos_filter.
13      Atomic increment of ringbuffer at pos_rb by filter
            value v_filter.
14    End
15  End
```

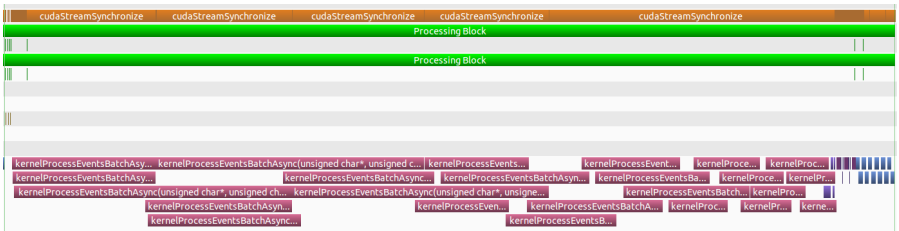
### 2.3.4 *Optimization 3*

The CUDA kernel itself is now highly optimized but there is still one thing to do. By default all kernels are queued on the GPU and executed in a series. This behavior blocks independent convolutions for different filters and ring buffers until the previous convolution is finished. However, the different convolutions are completely independent from each other and therefore the system uses another CUDA feature, called streams. By using streams, it is possible to group different kernels into concurrent execution paths and the GPU is able to process different kernels at once as displayed in Figure 7. This increases GPU utilization and the throughput of convolutions per second.

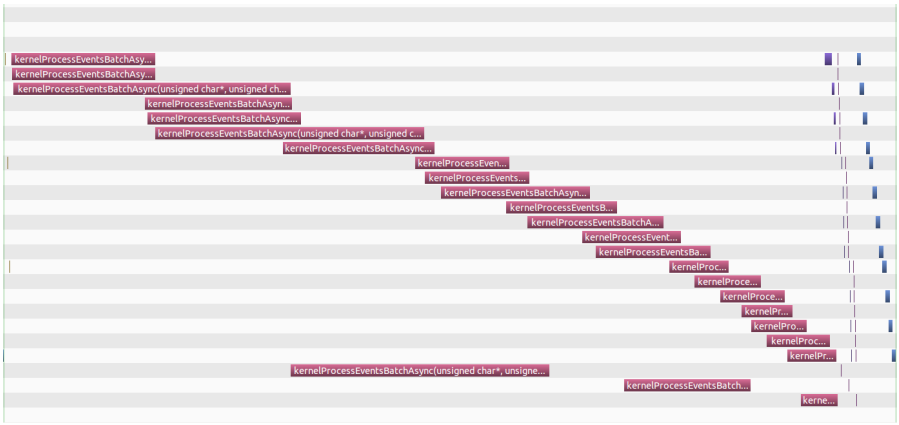
In the displayed output of Nvidia's visual profiler, the system processes a single time slot that contains the convolutions for 3 different filter speeds and 4 orientations each. This results in 24 convolutions to compute the output for all 12 combinations of speed and orientation. The plum bars shows the individual convolutions and the blueish bars at the end show the reading of the convolution result, the buffer resetting and the normalization of the motion energy. The system requires 13.7 milliseconds to process the 496, 261 and 140 events for the three different filter sets. In sum, this results in 21528 convolutions in 13.7 milliseconds which equals 1656000 convolutions per second. This computation is a bit cheating because the 496+261+140=897 events are basically the same for the different filter sets and the system is not able to process 900 individual events at the same time. Nevertheless, this calculation shows the power of the developed framework.

### 2.3.5 *Optic-flow estimation*

Computing the optic flow out of the filter responses can be done after computing the motion energy of the quadrature filter pairs as displayed in Equation 9. After normalizing the motion energy as described in Section 1.2, displayed in the center of Figure 3 and visible




(a)




(b)

Figure 7: Nvidia Visual Profiler Output of 24 convolutions per event with 3 speeds and 4 orientations each. The 3 different filter sets processed 496, 261 and 140 events for the current time slot in 13.7 ms. In Figure 7a, the plum bars above each other indicate convolutions that happened at the same time (up to 5). Figure 7b shows the different CUDA streams to group operations n concurrent execution paths.

in Figure 7 on the right side as blue bars, the filter responses indicate the probability of a moving edge's direction and speed.

As shown in Figure 3, the optic-flow is not computed in the convolution thread, because the computations done in this thread have to be as simple as possible to keep up with the incoming data. Additionally, the PUSHBOT controller output, generated to calculate the steering commands, is much rarer compared to the high frequent motion energy updates and therefore most of the computed optic flow would be unused. Hence, the optic flow computation is located in the PusBot controller thread itself.

In this implementation the resulting per-pixel optic-flow is computed by simply taking the direction and speed from the filter with the highest motion energy, in parallel for all pixels. After several failed attempts of interpolating the motion energy, speed and direction it has emerged that interpolation this kind of data is very difficult. Nevertheless, taking the maximum response gives reasonable good results and therefore and because of limited time, the interpolation problem was not further investigated.

To get rid of any noise in the events data and thus minor filter responses at random locations, a threshold is applied to the final per pixel motion energy: An energy threshold of 0.6 produces the optic flow in Figure 9 which is based on the motion energies shown in Figure 8.

## 2.4 OPTIC-FLOW-BASED ROBOT NAVIGATION

As addressed in the introduction, the navigation principals are based on the behavior of bees and the findings from [2]. When a bee travels through a corridor, her brain tries to keep the optic-flow, measured with its compound eyes, on the left and right eye equal. When the bee approaches an obstacle, the optic flow in this direction increases and the bee turns to the left or right to avoid a collision. This technique is very simple and computational inexpensive compared to the stereo vision of humans and therefore it is very easy to implement.

After computing the optic flow as described in the section above, it is possible to compute a steering output for the PUSHBOT based on the technique introduced just now. This is done by implementing a simple PID controller as shown in Equation 10.

$$\begin{aligned} e_{sum} &= e_{sum} + e \\ y &= K_p \cdot e + K_i * e_{sum} \cdot dt + K_d \cdot \frac{e - e_{old}}{dt} \\ e_{old} &= e \end{aligned} \tag{10}$$

The error signal $e$, used as input, is based on the estimated optic flow: In a first step, the system computes the average optic-flow on the left and right image half ($F_{L,avg}$, $F_{R,avg}$), weighted by the per pixel motion energy $E_{L,i}$ respectively $E_{R,i}$ to increase the influence of pixels with high motion energy. The final error signal $e$ is then computed by taking the difference between the left and right horizontal optic-flow as shown in Equation 14 weighted by the horizontal component of the motion energy ($|E_{L,x}|$ and $|E_{R,x}|$). The vertical component of the optic-flow is ignored, because the PUSHBOT can't react to an error in the vertical direction.

$$F_{L,avg} = \frac{\sum_i F_{L,i} \cdot E_{L,i}}{|E_L|}, F_{R,avg} = \frac{\sum_i F_{R,i} \cdot E_{R,i}}{|E_R|} \tag{11}$$

$$|E_{L,x}| = \sum_i \cos \alpha_i \cdot E_{L,i} \tag{12}$$

$$|E_{R,x}| = \sum_i \cos \alpha_i \cdot E_{R,i}, \alpha_i = \text{flow direction} \tag{13}$$

$$e = \frac{|E_{L,x}| \cdot F_{L,avg,x} - |E_{R,x}| \cdot F_{R,avg,x}}{|E_{L,x}| + |E_{R,x}|} \tag{14}$$

Afterwards, the output of the PID controller $y$ has to be mapped to the two motor speeds. This is done by taking an average speed $S_{avg}$ and subtracting and adding $\frac{y}{2}$ from the left respectively right motor as shown in Equation 15.

$$S_L = S_{avg} - \frac{y}{2} \qquad (15)$$
$$S_R = S_{avg} + \frac{y}{2}$$

A better approach and actually a goal when planning the system was to infer the average speed from the measured optic flow. This would allow the system to imitate another behavior of bees: They slow down their movement when the corridor gets narrow. This information could be extracted from the optic flow by measuring the average flow of the whole image. Smaller corridors produce higher image speeds than wider corridors. This technique is not yet implemented in the system because the estimated optic flow is not exact enough to produce such reliable data. This problem is described more in depth in Section 3.1.

The new computed motor speeds are transmitted 20 times a second to the PushBot to provide a smooth response to the constantly changing error signal.

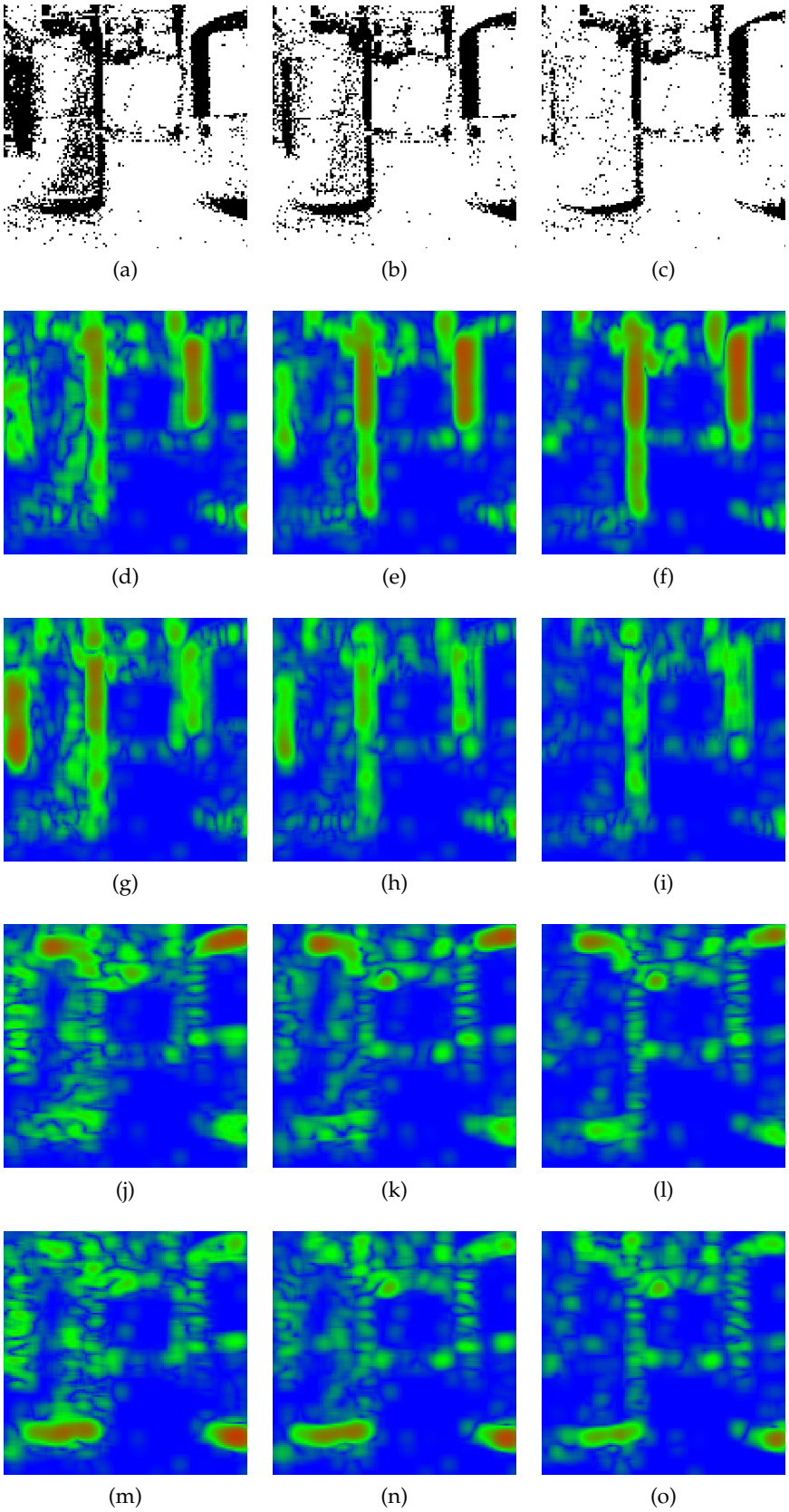
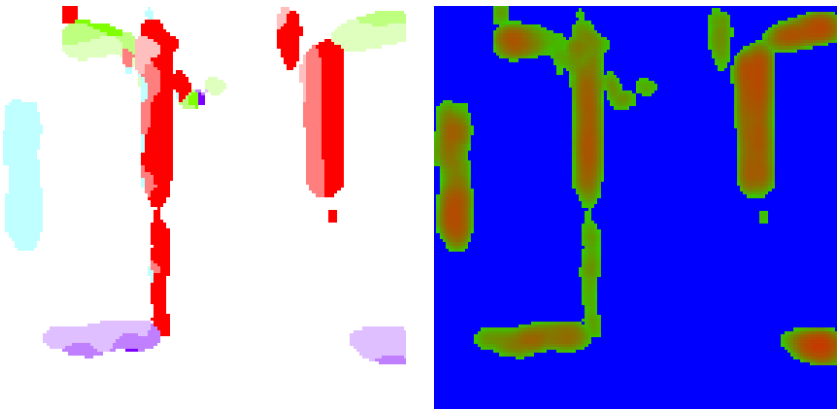



Figure 8: This figure shows an example frame with 3 different speeds (left to right column, 16.66, 33.33 and 66.66 px/sec), their events in the time window ($T_{window} = \{300ms, 200ms, 100ms\}$), 4 orientations (top to bottom, 0°, 180°, 90° and -90°) and their individual motion energy (blue: 0, green: 0.5 and red: 1.0)



(a) Computed optic-flow (b) optic-flow energy

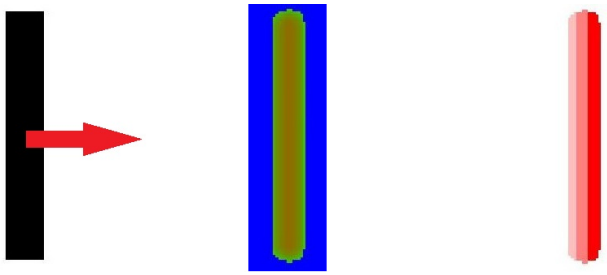
Figure 9: Shows the computed optic-flow and energy based on the motion energies from Figure 8. The optic-flow is colored by using the HSV color model: Hue indicates direction (0-360°), saturation indicates speed. Higher saturation means higher speed. Motion energy threshold: 0.6

# 3 RESULTS

After implementing the proposed framework, it was tested with the actual PushBot and the findings of this step are listed below. Due to several issues with the framework, the PushBot was not able to navigate very well through its environment. The issues are associated with the optic-flow estimation as well as the PID controller.

## 3.1 OPTIC-FLOW ESTIMATION

The implemented optic-flow estimation is able to reliably detect the direction of a motion but it has problems with detecting the correct motion speed. Figure 10 shows a single moving edge that travels from the left to the right with a constant speed of 33 $\frac{px}{sec}$. Instead of detecting a single speed, the implemented spatial-temporal filters (16, 33 and 66 $\frac{px}{sec}$) respond all to the same moving edge but with delayed responses. Therefore it is not possible to extract the correct motion speed. The displayed effect can be minimized by increasing the motion energy threshold but on the other hand this drastically decreases the optic-flow density. Furthermore, the system wouldn't be able to detect motions with similar speeds and it would have to process much more than three filter sets. Therefore, increasing the threshold is not a solution.

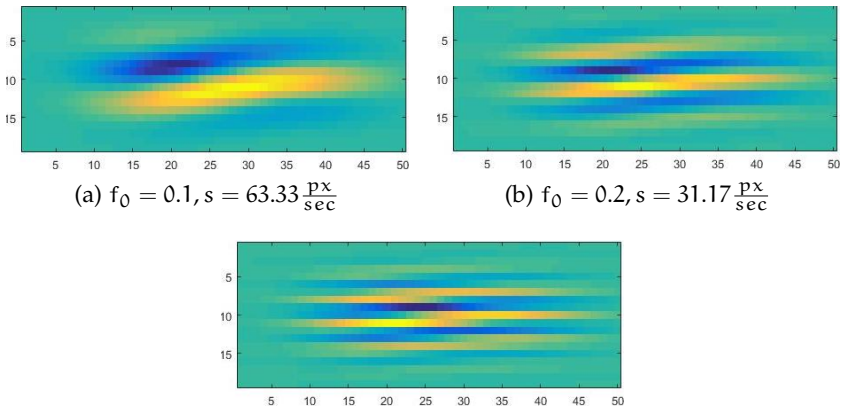


(a) Events in $T_{window}$ = 400ms (b) Motion energy for computed optic flow. (c) Computed optic flow.

Figure 10: Single moving edge with a speed of 33 $\frac{px}{sec}$. The systems uses filters sensitive for a speed of approx. 8, 16 and 33 $\frac{px}{sec}$. Motion energy threshold is set to 0.6. The figure shows that all three filters respond to the same moving edge with different delays (due to different time windows) therefore, the optic flow contains all three speeds.

This described issue mainly arises from the fact that the system allows different time windows to detect different speed but using equal

time windows introduces new problems: Figure 11 shows three filters with equal time windows. By changing their filter parameters, they are able to detect different speeds with the same time window. The downside of this approach is, that these filters change their spatial frequency when changing the filter speed and therefore they change their sensitivity for similar speeds. Additionally, it is not possible to reach very low speeds because of the limited spatial resolution. In Figure 11 the filters are constructed with a spatial and temporal resolution of $19 \times 19 \times 50$ and the slowest filter detects speeds of approximately 21 $\frac{px}{sec}$. As shown in Figure 11c, even with such a high spatial resolution, the blue and yellow regions are just one pixel wide and therefore it is not possible to decrease the preferred speed furthermore.



(a) $f_0 = 0.1, s = 63.33 \frac{px}{sec}$ (b) $f_0 = 0.2, s = 31.17 \frac{px}{sec}$

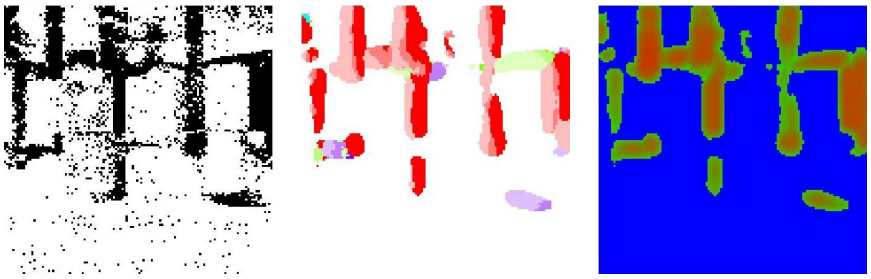
(c) $f_0 = 0.3, s = 21.11 \frac{px}{sec}$

Figure 11: This figure shows three spatial-temporal filters (XT-slice) with an equal time window but different speed sensitivities. $\mu_{bil} = 0.23$ and $T_{window} = 150ms$ for all filters. By changing $f_0$ not only their preferred speed but also their sensitivity for similar speeds changes.

Due to the described problems above, detecting the exact motion speed was one of the main problems. The images in Figure 12 show an example scene where the PushBot is moving towards cylindric obstacles and a door in the distance. It is clearly visible that the motion direction is consistent but the speed detection lacks of distinguishable information. The whole scene moves with the same speed according to the optic-flow saturation in Figure 12. This is either caused by a wrong filter setup were the visible motion is too slow to be detected or it is caused by different filter time windows as described above.

Detecting even slower movements is very hard, because it is necessary to increase the time window up to 500 ms and therefore, the motion must last even longer to generate a valid filter response. In addition , increasing the time window also increases the delay between motion and filter response.

Designing multiple filters, sensitive for similar preferred speeds (necessary to distinguish between the slow moving fore- and background) causes supplemental problems. As shown in Figure 11b, a filter is always sensitive for similar speeds due to its spatial spread and therefore multiple filters with closely spaced, preferred speeds would response to a moving edge with very similar or almost equal motion energy. Furthermore, it is not possible to decrease the preferred filter speed endlessly because of the limited spatial and temporal filter resolution: The minor speed differences are not representable anymore as shown in Figure 11c even though the spatial resolution is still twice as high compared to the filters from the introduction (Figure 1).



(a) Events in $T_{window} = 300ms$.

(b) Computed optic flow colored with HSV color model: Hue indicates flow direction, saturation indicates speed.

(c) Motion energy measured by filter with highest response (per pixel). Energy threshold = 0.6

Figure 12: Shows an example scene were almost every moving edge has the same speed. The systems uses filters sensitive for a speed of approx. 8, 16 and 32 $\frac{px}{sec}$. Motion energy threshold is set to 0.6.

## 3.2 ROBOT NAVIGATION

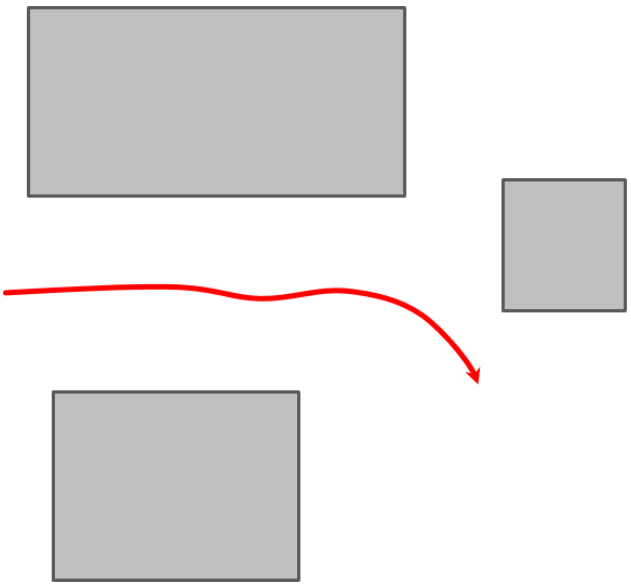
As described in Section 2.4, the control loop takes the optic-flow as input and computes the error signal based on the difference between the left and right image speed. Therefore the quality of the error signal highly depends on the computed optic-flow. Due to the problems with detecting the actual motion speed, the error signal is almost binary because it is only related to the movement direction (left or right) and barely to the image speed. However, the PID controller used to compute the control output needs a proportional error signal to work properly. This is not the case and therefore it is very hard to adjust the PID parameters to get a smooth movement.

Despite the arised problems, the system is able to navigate in very simplified environments. Figure 13 shows the result of a small sequence with the PID parameters set to $K_p = 0.2$, $K_i = 0.3$ and $K_d = 0.005$. Figure 13a displays a simple sketch of the top view on the

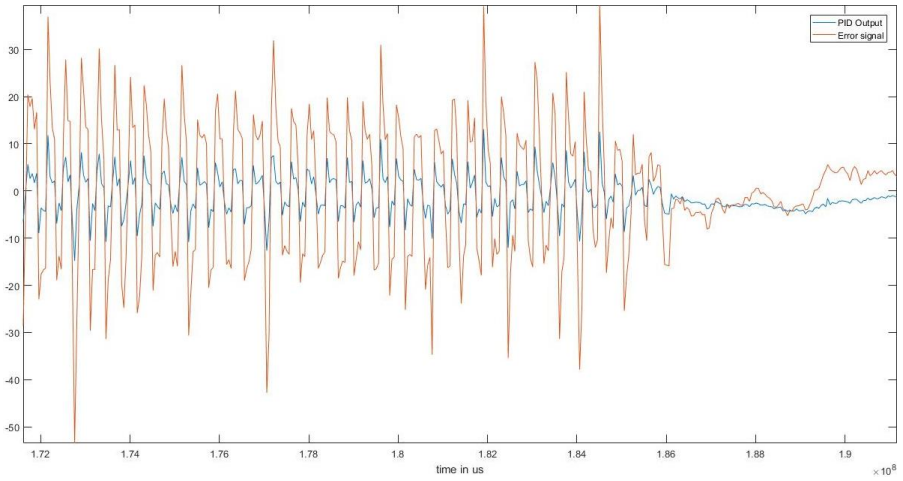
scene and the basic movement of the PushBot. Figure 13b shows the actual PID output (blue) and the error signal (orange) used as input. It is clearly visible that the PID oscillates around the zero crossing, therefore the PushBot turns alternatively to the left and right. This generates an opposite optic flow and the consequential error signal. This could be fixed by further adjusting the PID parameters especially by decreasing the proportional coefficient. However, this also decreases the sensitivity of the PID controller.

The end of the sequence shows a more constant error signal and PID output. This is caused by approaching the obstacle in front of the PushBot and the free space to its right. The optic flow on the right is almost zero and therefore the PushBot turns to the right to avoid the faster moving region. After avoiding the obstacle, the error signal gets positive again, the PID output converges towards zero and the PushBot moves in a straight line.

As a summary, the PID controller itself works as expected and is able to guide the PushBot, even with an imprecise error signal, through this very simple environment.



(a) Draft of a simple scene.




(b) Output of PID controller (blue) and generated error signal (orange).

Figure 13: Draft of a simple scene with an approximate length of 7 meter. 13b shows the actual error signal and the output of the PID controller.

# 4

## SUMMARY

This works shows an example implementation of a optic-flow based robot navigation system that uses very simple, biologically inspired mechanisms to navigate through its environment. It takes an optic-flow as input and tries to navigate through a corridor according to the navigation principals of bees.

The implemented framework computes the optic-flow, based on the sparse event stream, generated by a DVS which is integrated in the PushBot platform. Spatial-temporal filters, described in [1] are used to estimate the optic-flow, present in the recorded scene. This optic-flow estimation is used as input for the navigation planning step, were a simple PID controller tries to avoid fast moving image regions. It computes its error signal by taking the difference between the average optic-flow on the left and right image half. This error signal, used as input for the PID controller, generates a control output that is afterwards mapped to the left and right motor speed to initiate turning maneuvers, if necessary. This allows the PushBot (in theory) to navigate through the scene without colliding with any walls or obstacles.

The implemented optic-flow estimation, accelerated by CUDA, works very well but has its problems with detecting the correct motion speed, which is a very important information for any further processing step. Additionally, in this setup the system has to detect multiple similar and very slow movements which doesn't work as precise as expected. Despite from the issues with incorrect motion speeds, the PID control loop works as expected and the PushBot is able to avoid regions with high image motion in a very simplified setup.

Unfortunately, the estimated optic-flow was not precise enough to generate an accurate error signal for the PID controller. Therefore the PushBot was not able to distinguish between fore- and background as good as expected. Most of the time, he drives straight into an obstacle.

There are several topics, were the system could be optimized furthermore. The system was tested on a middle-class laptop with an outdated graphics card and therefore it would highly increase the throughput of events when the system runs on a high-end workstation instead. Another optimization step would be the design of more suitable spatial-temporal filters as described in the last chapter. And finally it would be appropriate to replace the PID controller with a more complex system such as an Kalman filter to produce a better control output even with noisy and not very accurate input data. The

PushBot also offers an integrated inertial measurement unit (IMU) to monitor its current orientation and rotation speeds. This additional information could also be processed by the Kalman filter and adds the knowledge about the current rotation speed and orientation. Furthermore the used default speed $S_{avg}$ show in Equation 15 could be replaced by a dynamically computed average speed based on the speed of the moving scenes to automatically slow down movements when the PushBot approaches a narrow gap.

Nevertheless, the whole project was very interesting and it greatly improved my skills in utilizing the GPU as powerful computing platform. It also taught me the very interesting principals in detecting motions with a filter-based approach.

# 5 APPENDIX

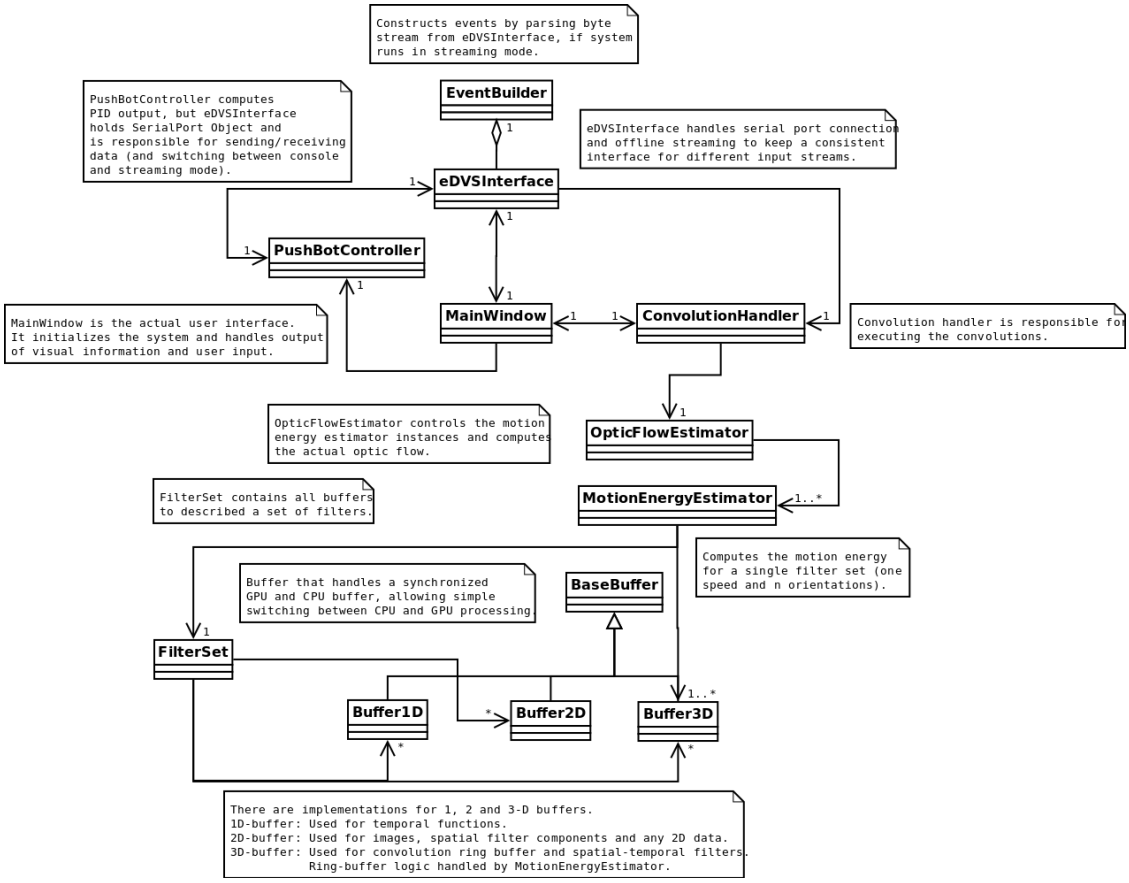



Figure 14: Additional view on the described system: An UML class diagram that shows the basic components of the system and their individual relations.

# BIBLIOGRAPHY

[1] Tobias Brosch, Stephan Tschechne, and Heiko Neumann. "On event-based optical flow detection." In: *Frontiers in neuroscience* 9 (2015), p. 137.

[2] Mandyam V Srinivasan. "Honeybees as a model for the study of visually guided flight, navigation, and biologically inspired robotics." In: *Physiological reviews* 91.2 (2011), pp. 413–460.

[3] Edward H Adelson and James R Bergen. "Spatiotemporal energy models for the perception of motion." In: *JOSA A* 2.2 (1985), pp. 284–299.

## COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

*Final Version* as of March 19, 2017 (`classicthesis` Version 1.0).