Master Project

IP712

Parallel Computer Vision:

Person Data Extraction

Lang Christian

Student number: 08-169-047

Fachhochschule Nordwestschweiz

Advisor:

Prof. Dr. Christoph Stamm, FHNW

Closing date: 27.01.2013

Abstract

Face recognition has been established in many environments these days. It is used in security systems, social media platforms or in digital cameras to support the user. In addition, the rapidly rising number of CPU cores in modern PCs or handhelds let us do more complex work on a single machine.

The central question of this work is: Is it possible to create a system that can detect and recognises people in a video stream in real time and only with the resources of one PC, but with at least one GPGPU capable graphics adapter?

To answer this question, such an application is developed with the use of C++, the computer vision library OpenCV and the GPGPU language CUDA from nVidia. To optimize the application for real time usage, the Concurrency Visualizer of Microsoft Visual Studio 2012 has been used. It is shown how to use difference images to calculate motion in videos and how to stabilize such motion areas with the use of a self-designed sweep line algorithm.

In the first part of this master project, the technologies to create such software are evaluated and the first steps of video processing, motion detection and speed optimization are done.

Table of contents

1 Introduction

1.1 Motivation

Face recognition is a very often used technique today, to add useful information to images where people are shown. For example to tag the birthday photo album with the names of each person around the cake, to detect all smiling faces in our camera viewfinder and automatically take the shot or to support criminal investigations with automatic filtering of big suspect databases.

But only still images are processed in all these use cases, even if these images are extracted from a video source, which includes much more information than a single picture. There are several images of the scene within a second, which could be used to create a more detailed image as with the extraction of only one frame of the image. The motion of people could be used to detect where a person is located in the scene or even to recognize a type of walking, to categorize old, young or handicapped people. If this thought will be taken much further, even the sound or speech could be used to recognize people.

To extract and use all this information, there are new techniques and algorithms needed than the known one of still image processing. However, the use of the available hardware has to be extended also, to be capable of processing the higher amount of data from a video stream.

To handle this much bigger effort, all available processing power of a modern personal computer (PC) should be used. This leads to techniques of parallelization and “General Purpose Computation on GPU” (GPGPU). Whereas parallelization ensures that the growing number of cores in a single CPU is fully loaded all the time, GPGPU enables the user to doing not only graphic calculation but also all other kinds of processing on all GPUs of his graphics adapter. Whereas a CPU core executes a single instruction on just one data block (Single Instruction Single Data, SISD), GPUs are capable of processing multiple data with the same instruction in parallel (Single Instruction Multiple Data, SIMD). This big advantage of GPUs should be used.

1.2 Task definition

A sample application should be developed in this master project, to show that modern PCs are capable of processing a live video stream to extract person data and recognize already known people therein. To specify the desired use, the application of a security camera has been taken, which is not moving and provides video streams of hallways or stairwells. This stream should be processed in real time and should deliver position and orientation of people in the scene and the best matched person in the database. A detailed description of the overall project can be found in the project clarification in appendix A.

The goal of Project 7 is the ability to detect each human body in the scene of a video. The video streams for testing has been recorded, so they can be played as needed. The video should be processed in real time. But because of the fact that the further steps of person recognition will use much calculation power too, the part of person detection should be thrifty with the available calculation resources. So there should be simple and fast algorithms to be chosen to make the detection as light and fast as possible.

The project should use the computer vision framework OpenCV with the C++ interface because this framework already contains many algorithms that are needed. Also the Microsoft framework AMP and the corresponding C++11 standard should be used to gain knowledge about these two new technologies. In addition nVidia’s CUDA should cooperate with OpenCV and provide support for GPGPU computation.

2 Test Data and Environment

This chapter describes the collecting of test videos and the used development and test environment.

2.1 Collecting Test Videos

Some test videos with defined events are collected which can be used for testing the software. A detailed plan of the scenes and places has been created. This was needed to ensure that not too much time is used for recording different events with a camera. The plan is shown in Appendix B and contains all information about people in the scenes, how they behave and what they are wearing. The plan is also listing how many different people are shown in one view and how they move through the scene. The places where the scenes are recorded are also defined.

Although there is such a detailed plan, some of the prepared scenes could not been recorded. Especially no person with dark skin could be found who are willing to help for the test data. So those scenes could be recorded in a later state of the project, when the software will be tested with harder situations.

These videos had been recorder, because the software has to process a video stream in real time. So there must be a source of such a stream. If it is from a security cam or from a video file on the hard drive does not matter. At the time of development a live stream from a camera is not useful because in this case, some test scenarios with defined events cannot happen each time when the software has to be tested.

2.2 Meta Data for Testing

It would be handy at a later time, if there would be a possibility to easily write automated tests of the application. For this purpose some additional metadata of each video should be available. Automated tests could then evaluate, if a detection of a person should actually happen at the processed scene. This could be realised with tags which describes how many persons are visible in the image at each timestamp.

To do such a tagging, the class `people::TestVideo` is prepared for reading in a CSS-file for each video file. The tags and the CSS-files had to be generated at a later time, if they are needed.

2.3 Hardware Environment

The PC hardware in Table 1 has been used for coding and testing.

PC FHNW Number	Specifications
st10i73061	Mainboard: ASUS Rampage II Extreme CPU: Intel Core i7 950 3.06 GHz GPU: ZOTAC GTX-460 AMP 1GB DDR5 RAM: Mushkin Redline 3x2GB DDR3-1600 Hard drives: Samsung 2x SATA 500GB

Table 1 - Used Hardware Equipment

3 Detection of the Human Head

As the project clarification in appendix A describes the goal of the Project 7 is to detect human heads in a movie. The location and the orientation of the head should then be used to create a model out of the detected images, with which a person can be recognised in another scene. There are several variants to do such a head detection.

A function simple to use, which is offered by OpenCV, is the class `cv::gpu::HOGDescriptor`, where HOG stands for Histogram of Gradients. It uses multiple scales of the source image to detect predefined patterns of pedestrians. The result is an array of rectangles in the image where the detector has found fully shown people. The mentioned version of the descriptor runs on GPU, so it is very fast in comparison to the one on CPU. Nevertheless it needs much calculation power to process a full image. There are also no pre-trained descriptors for heads or eyes, only for the whole body.

Another OpenCV function is the class `cv::gpu::CascadeClassifier`, which uses HAAR like features on cascaded images. This method has many pre-trained feature sets, for example: eye, face or body. This function can be additionally executed on the GPU and as it uses also the approach with multiple different scales of an image, therefore has similar performance as the HOG descriptor. Because of the fact that this variant is good in detecting eyes or faces, it could be used in a later state, where the ROI of the heads has already been detected.

Also another approach is to subtract the background of the scene and retrieve all the foreground objects. In this project an even simpler approach is chosen, to achieve the best performance. The used motion detection is described in this chapter.

3.1 Region of Motion

To recognize the human body, a good reduction of the problem is to detect moving parts in the picture. Because of the fact that a human being is always a little bit in motion, this approximation is a good approach. On the other hand, it is only useful in situations like ours where the camera stands still, because a moving camera looks like that all parts of the picture are moving. The area that is extracted by a detection function is called a “Region of Interest” (ROI) and will be calculated like described in this chapter. The code for this functionality is contained in the class `people::MotionDetector`.

3.1.1 Image scaling

To reduce calculation cost of the motion area, the image is resized to a height of 200 pixels. With this size the calculation duration can be dramatically reduced and the detection rate of motion is good enough to detect a walking person. Since the image size has been reduced, there will be fewer pixels that have to be processed in further steps. Figure 1 and Figure 2 show the difference between with and without scaling. The whole process of motion detection takes about 40 milliseconds on the original size, whereas the processing of the resized frame takes only about 2 milliseconds.

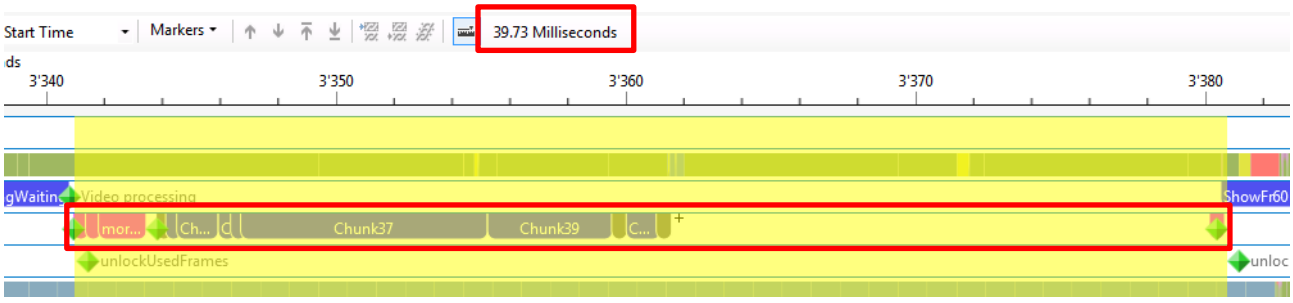


Figure 1 – Motion Detection without resize (Height: 720 pixels)

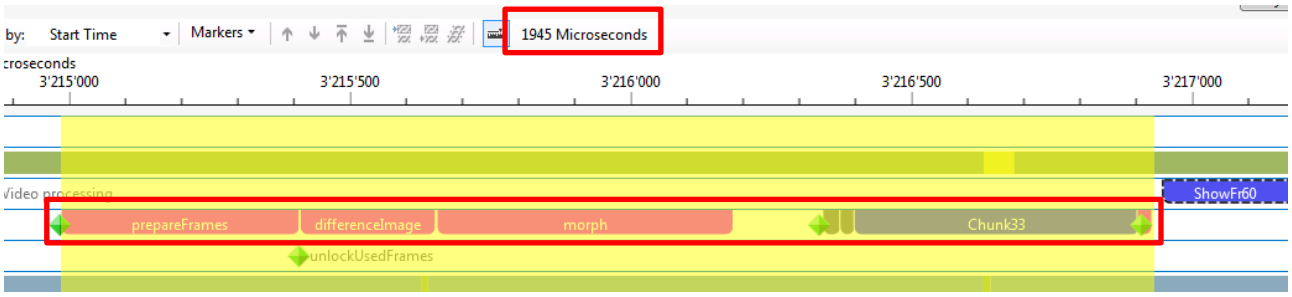


Figure 2 – Motion Detection with resize (Height: 200 pixels)

3.1.2 Calculation of motion

To calculate the motion in video, an image is subtracted by its predecessor image of the video stream. This results in an image, in which bright points symbolize big changes in this pixels and indicates a motion. Due to the fact, that the value of a pixel that has changed is not important, the picture is been threshold to receive a binary image. After that, all moved pixels are now white and all areas that are not moved remains black in the image.

Figure 3 - Binary Image of Motion Detection

Some noisy pixels can appear, as highlighted in Figure 3, because there are always some changes in lighting conditions and noise in the image-sensor of the camera. This noise is removed by a morphologic close operation.

3.1.3 Region detection with clustering

The gained motion pixels in the image symbolize the area of motion. Now the problem is that not each pixel is a single ROI, further many pixels together build a ROI. For this reason, all found points had to be clustered, so that at the end, several groups of pixels results, in which all points belong to the same object.

A solution for clustering could be the K-Means algorithm. This is a very fast algorithm with one big negative property. It needs the desired number of clusters as input. Because there is no chance to know how many people or moving objects are visible in the image, this is no solution.

The hierarchical approach matches the problem better. It does the clustering as it calculates the distances between all elements and groups those who are close enough. To do this, there is a function in OpenCV,

which performs such a clustering with a distance predicate given. This predicate decides, if two points belong to the same cluster or not. The function `cv::partition` [1] has the following signature.

```
template<typename _Tp, class _EqPredicate> int partition(const vector<_Tp>& vec,
vector<int>& labels, _EqPredicate predicate=_EqPredicate())
```

The predicate, which has to be provided, has been implemented to calculate the Manhattan Distance and decides, if the two points belong to the same group. The Manhattan Distance has been chosen because of its simplicity and thus fast execution. To define the minimal distance between two points, a model has to be defined. The model has to show the different ratios between the image size, the desired minimal person size and all that in the union of pixels. The only restriction that has been defined is that all bodies which minimum are half the height of the image, should been detected. Figure 4 shows the used ratios. With the image height of 200 pixels, a minimum clustering distance of 25 pixels has been defined, which corresponds to the height of the head.

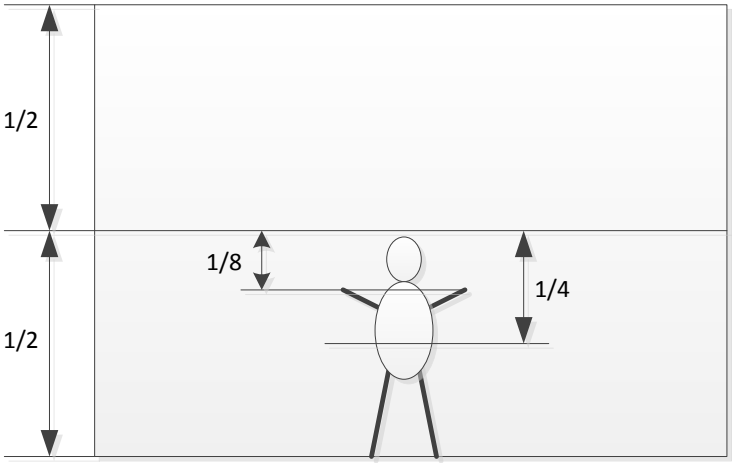


Figure 4 - Ratio of the minimal body and the frame size

Further, there has been a minimum size of a detected ROI defined. This is approximately the half body height which has been chosen as 40 pixels. The corresponding ROI width has been chosen as 10 pixels, which is justified with Figure 5. It is to note, that all these ratios depends on a very basic approximation of the size of a human body.

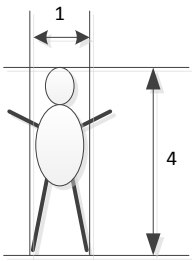


Figure 5 - Ratio of the Human Body

Clustering of many points needs much time because the algorithm has to calculate the distance of every point-pair. If n points exist, this results in n^2 pairs. Because the `cv::partition` function runs only on one core, a straight-forward solution is to split the whole set into chunks and to do the clustering for each chunk on a separate core. Since the motion is always concentrated on several points, there will be chunks that will take longer for clustering then others. The have to be small to use the capacity of all cores, but not too small to generate too much overhead for scheduling. Figure 6 shows the used 64 chunks for clustering.

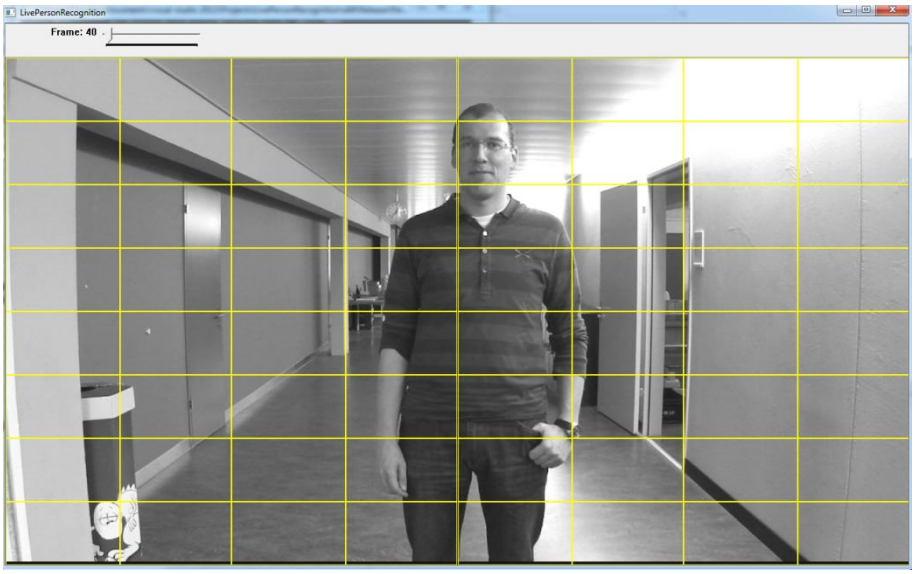


Figure 6 - Separation of Image in Chunks

The parallelization has been implemented with a simple `parallel_for` of OpenMP, which creates sufficient load for all four used cores by our system. The calculation of all chunks of a single image is shown by the blue and grey bars in Figure 7.

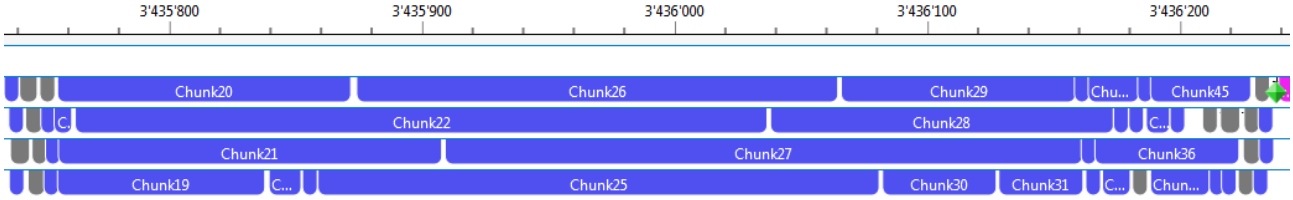


Figure 7 - Processing of Chunks

3.1.4 Union of separated ROIs

To union the separated ROIs and gain the ROIs over the whole image, they have also to be clustered. Because the number of final ROIs is also unknown in this case, the hierarchical clustering is used too. This process is done on one core. Some of the sub regions, before the union, are shown in Figure 8.

Figure 8 - Sub Regions of Motion Detection

3.2 Stabilize Motion Region

The previous described detection of motion regions is a very fast and simple way that has been found. But because these regions are newly calculated after each frame, they change very fast. Therefore, even small regions, which can be triggered by some short reflections in the picture, can interfere with the actual required region. As a matter of fact, this interference happens typically only at a few directly following frames. Because it is necessary to suppress such disorders, this property can be used.

An example of such interference is shown in Figure 9. There are three detected motion regions symbolized with the colours: blue, purple and green. The red rectangle would be the preferred region, which frames the person in the image.

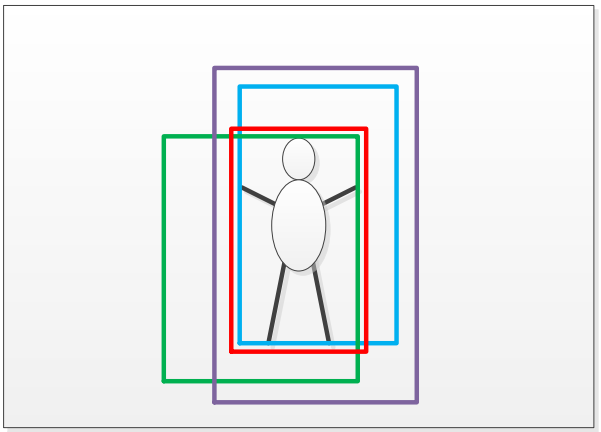


Figure 9 - Interference of Motion Detection

An approach to get this red region, is to calculate a “mean region” over the others. With this expression, the steady part of all involved regions is meant. This uses the knowledge that a human being moves always a little bit and do not rapid changes in position, like the reflections or other interferences do. So it is certain that a region, which is covered by all disturbed regions, contains the required area.

Figure 10 shows a simplified example for two interference areas and the desired red final region. As it can easily be seen, the red region is the intersection of the two others.

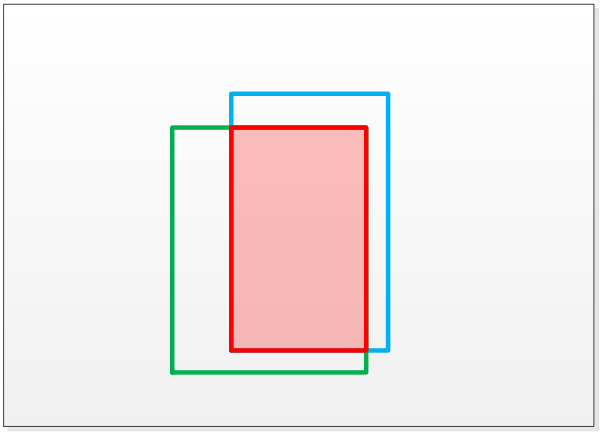


Figure 10 - Calculation of "Mean Region"

In the real application, it is preferred to calculate a “mean region” over many frames and select all regions, which have a minimum number of frames that intersects in this area. For example, 10 frames could be taken and a minimum number of 5, so all regions that have at least 5 frames, who covers that region, will be accepted. With this method, some little interruptions of the body motion could be compensated.

3.2.1 Simple Approach with Intersections

To calculate this minimum cover area, several approaches can be used. If the approach of calculating intersections is expanded, it can also be used for determine areas with minimum cover. A single intersection of each input area with each other would result in all regions that have at minimum 2 frames that covers it. For a minimum of 3 layers, the resulted areas could be intersected among each other and so on. This version can be easily confirmed, but unfortunately very expensive. To calculate the areas of a minimum l layers, there has to be $l-1$ iterations of all intersections where the calculation of all intersections results in an effort of n^2 operations and resulted areas, where n is the number of input areas. The resulting cost is shown in Equation 1.

$$O\left(n^{2^{l-1}}\right)$$

Equation 1 - Cost of Mean Region with simple Intersection Approach

As it can be reviewed, this algorithm is exponential in time and space. Therefore a l of 2 could be very fast, but already a l of 4 is much slower. A measure of a $l = 4$ is shown in Figure 11. The red line symbolizes the start of the `calcMeanMotionRegionIntersection` function in the class `people::MotionDetector`. At each green diamond an iteration, with the number of input areas shown in the box, is started. As it can be seen, the first start calculates 11^2 rectangles, the second 49^2 and the last one 903^2. The difference between each theoretical input and real input numbers stems from some optimization between each iteration.

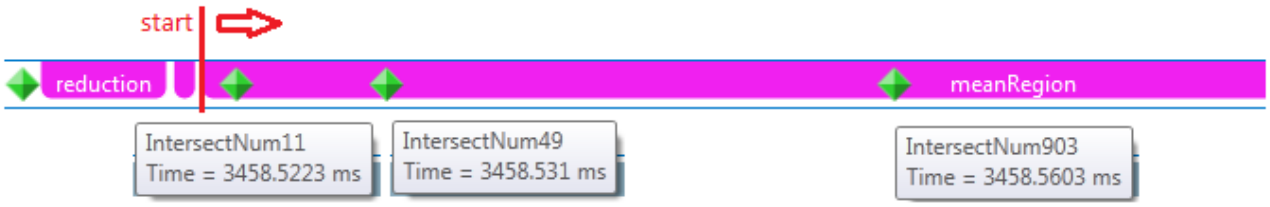


Figure 11 - Measure of simple Intersection Approach

This example takes about 400 milliseconds to calculate the resulting regions and is therefore not useful.

3.2.2 Approach of Sweep Line Algorithm

Because of the fact that this problem is in the two-dimensional space, an algorithm can be developed which uses the principle of a “sweep line” [2]. Such an algorithm usually has the advantage that it could be build up to have a typical time complexity of something about $O(n\ log\ n)$.

A data model has to be defined to develop such an algorithm. As shown in Figure 12, the sweep line is moved over the x-axis and each vertical edge of the rectangles is an event. Those events are collected in a sorted set and can be processed in the direction of the sweep line. The set itself is implemented as a binary search tree and is found in the C++ “Standard Template Library” (STL). Because these events depend on the rectangles, which are not changed during the algorithm, the “event queue” can be filled at the beginning.

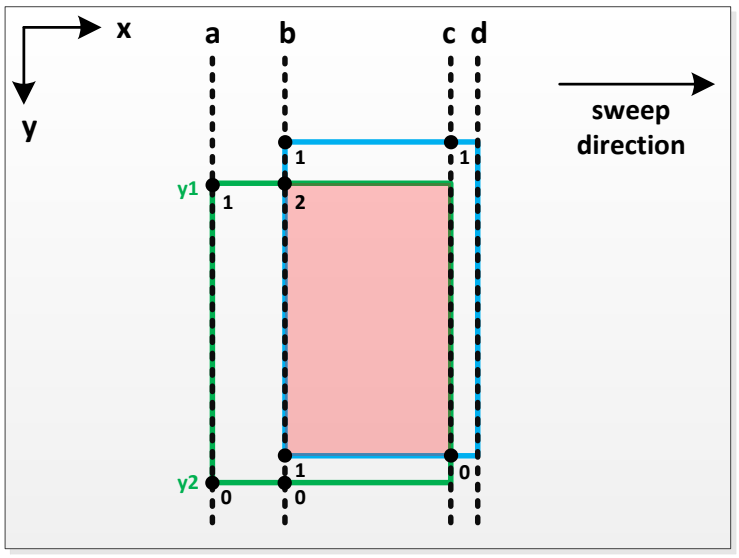


Figure 12 - Model of Sweep Line Approach

Processing of an event leads to a change in the state of the algorithm. The state itself is represented by a sorted set of points at the y-axis, where each point has a count of current underlying layers, and the current and last x-position. In Figure 12 the dotted lines (a, b, c, d) symbolize such states. For instance, the two points y1 and y2 belong to the state "a". This example has two input areas and should deliver all areas which has minimal layer coverage of 2. The number besides the points is the number of underlying layers after this y-position to the next point. Point y1 has therefore a $l = 1$, since the first rectangle begins, and point y2 has a $l = 0$, because the rectangle ends and no other rectangle lies beneath.

When the sweep line jumps to the next x-position, it is certain that all points, which have a $l \geq minLayer$, describe a required result area. In Figure 12 the point with $l = 2$ on state "b" is such a point. It is defined by the y-parameter and the last x-parameter. The current x-coordinate, in this case of state "c", would define the second x-parameter. To receive the second y-parameter, the sweep state, in this case state "b", will be searched for the next y-point which does not satisfy the $l \geq minLayer$ predicate. With this method, all generated rectangles lie between two sweep line states. This could be improved with an additional variable, which would memories the x-beginning of a satisfying rectangle and generates it, only if the last x-coordinate of it has been detected.

To process the sweep line states, two scenarios have to be considered. The first is when an event is an edge that starts a rectangle; second event is an edge that ends a rectangle.

Figure 13 shows the procedure, if a new line has to be added to the state. If the beginning point, named y1, is not already on the state it is been added. The layer count of all points between y1 and the ending point, named y2, of the current state are incremented. If already a point with the y-coordinate of y2 exists in the state, its layer count is not touched. The new y1 gets the layer count 1, if it is the first point of the state or the increment of the layer count of the previous point in the state. If the point has not been added because at this y-coordinate already a point existed, the layer count of this point is incremented.

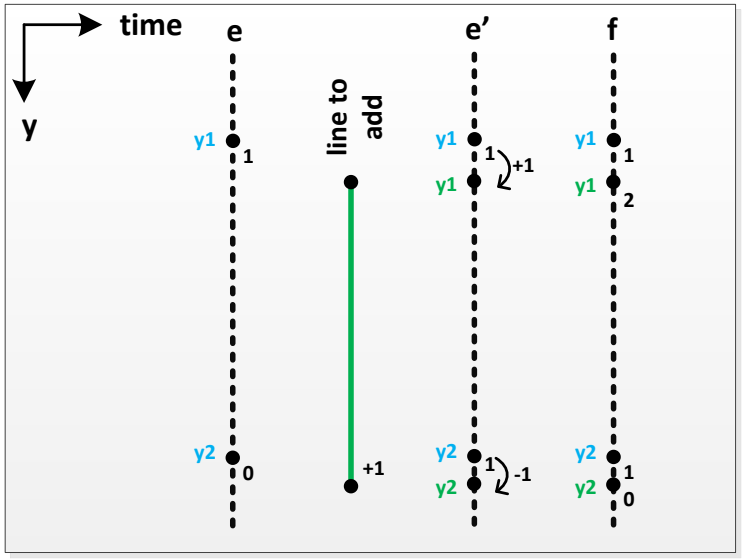


Figure 13 - Add new Line to Sweep State

The new point y2 always gets the decrement of the previous point, only if there was already a point with this y-coordinate, nothing is changed. This process has to be done with all new lines of the current x-coordinate, which are stored in the event queue.

Figure 14 shows the process to remove a line when a rectangle is ending. In this case, all layer counts of the points on the line, inclusive y1 but without y2, are decremented. After this step, it has to be determined, if the points are no longer used and can be erased. This is done by compare the current layer count of the points with the count of the corresponding predecessors. If they have the same count, the old points can be removed.

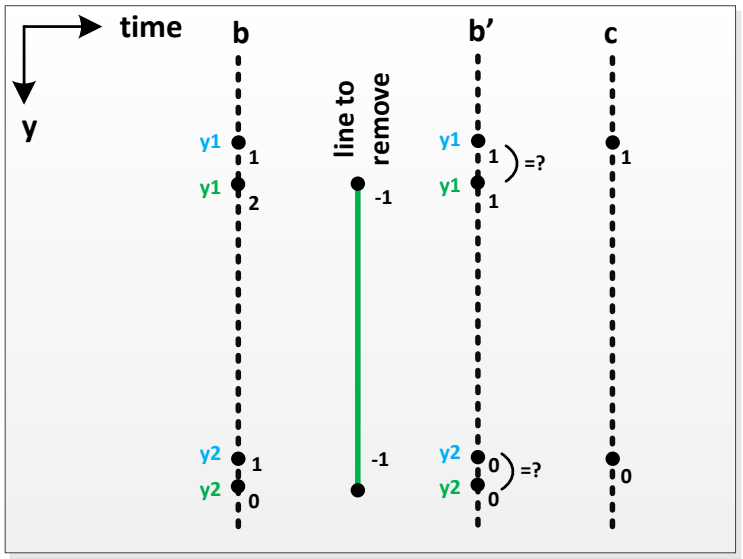


Figure 14 - Remove Line from Sweep State

The calculation complexity consists of several processes:

- Initialization of event queue: $O(n \log n)$, where n is the number of rectangles
- Iterate through event queue: $O(n)$
- Generate rectangles in current state: $O(s)$, where s is the number of points in the state
- Process `addLine` or `removeLine` : $O(2 * log\, n)$

Equation 2 shows overall time complexity. The term s is special, because it depends on how many lines of the event queue are in the state at any given time. Because s could go to n, the resulting complexity is quadratic.

$$O(n \log n + n * (s + log\, n)) = O(n * s + n \log n) = O(n^2)$$

Equation 2 - Cost of Mean Region with Sweep Line

This is not really good, but much better than the exponential complexity of the presented simple approach (Equation 1). Further, there could be an improvement to reduce the time of generating the rectangles.

The described algorithm is implemented in the function `calcMeanMotionRegionSweepLine` in the class `people::MotionDetector`.

3.3 Head and Orientation of Head

The next step is to detect the head in the found region of motion, where a person is almost certain located. There are several approaches to detect a head. One possible solution is to do a pattern search for head specific features, such as: eyes, ears or nose. This can also be used to determine the orientation of the head. If both eyes, the nose and the mouth is found, it is certain that the person looks in the direction to the camera. If only one ear and possibly the nose are found, the person looks in the direction of the vertex from the ear to the nose. And if both ears are detected but nothing else, the person looks away from the camera.

This detection can be implemented with the already mentioned `cv::gpu::CascadeClassifier`. But because this method uses many rescaled images of the search region and has to match all mentioned patterns over this many versions of the image, it would be slow. Therefore a system is needed which uses the known motion region to calculate the area of the head.

A possible approach could be to calculate the position of the head only with the knowledge, that if a person stands completely in the picture, the head should be in the upper middle of the detected region. This would look like it is showed in Figure 15.

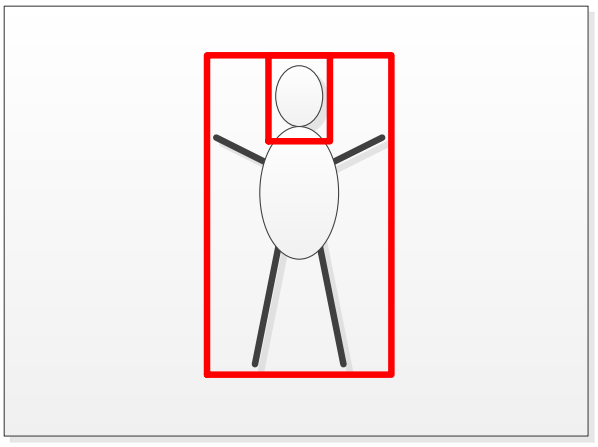
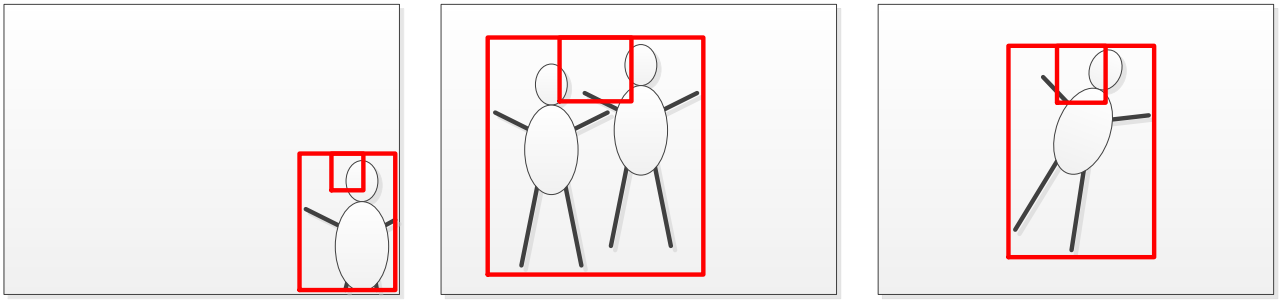


Figure 15 - Optimal Case for Head Detection

Unfortunately this is not often the case. Situations like in Figure 16 are also plausible and do not deliver the expected results.



a b c

Figure 16 - Other possible Cases for Head Detection

Especially the case “b” in Figure 16 could be tricky to handle, because knowledge of how many people produce the detected motion region is needed in such situations. This could only be gathered, if the two persons produce each a separated motion region first. At the moment, when these two regions merge to one single region, the knowledge that this region consist two persons is obtained. This knowledge could also be useful for other approaches of head detection.

To solve the other problems, an optimized version of the `cv::gpu::CascadeClassifier` could be used. For instance, to search for the number of heads only, as it is known to be in the region and abort, if the defined number is found. The already tested approaches can be found in the class `people::HeadDetector`.

3.4 Head Model

The head model is the used data model which stores all information of a known person’s head. This model has not been developed yet. A possible solution is shown in the project clarification in the Appendix A part 5.d.

This proposal has similarities with the model of the project TLD-Vision [3] which is presented on YouTube [4]. It seems that it uses some movement prediction to track the object and collect pictures of the object in different orientations and distances. The collecting of a new picture could be triggered, when the difference image between the already saved images and the current image is too big. The collected images build the model of the object together, which can be used to detect the object, if it was lost for some reason. It could be interesting to investigate the source code, which is licensed under GPL with the name OpenTLD [5], because this algorithm seems to work really great.

4 Implementation

The mentioned steps in chapter 3 had to be implemented in Project 7. This chapter shows the different libraries and frameworks that are used and also others, which have been evaluated but have not satisfied the requirements. The used libraries are shortly described and specialities shown.

4.1 OpenCV

OpenCV is a big open source image processing library. It offers many useful functions of low- to high-level applications. To master this huge amount of possibilities and powerful modules, this chapter describes several specialities of this library.

4.1.1 Compilation

To get a fully compiled version of OpenCV inclusively the source code and the debugging symbols, we have to compile it by ourselves. The prebuild version of OpenCV, which is available on SourceForge, is stable and correct compiled and can be used as a base version. If newer features are needed, we can use the self-compiled version, but with the handicap, that it can be a buggy version. A tutorial for self-compilation is available on the OpenCV Wiki [6].

4.1.2 GPU Support

OpenCV has ported many functions to CUDA, to use the massive acceleration power of a GPU. The corresponding libraries are already included in the precompiled version of OpenCV. These two libraries are found in the following subfolder: `opencv\build\gpu`.

4.1.3 Visual Studio 2012 and GPU Support

The new AMP framework and the optimized C++11 standard are two reasons to have Visual Studio 2012 support within a C++ project. Because of the fact that the precompiled version of OpenCV only supports the old Visual Studio 2010 Platform Toolkit, the only solution is to compile the whole OpenCV from scratch with the new vc11 compiler. The current version on GIT (2.4.9) has already support for CUDA 5.0 and the Visual Studio 2012 Platform Toolkit. The only problem is the CUDA compiler nvcc, which is not ready for the new Toolkit. There are several solutions on the web [7] [8] which offer a workaround for this problem. Many of them only try to tell the wrong version of the Visual Studio Compiler to the nvcc. The target file of the NVIDIA Build Customization has to be edited for this purpose. To enable the CUDA 5.0 Toolkit to work, the changes of point 2 on the NortuRE-Blog [8] can be applied to the files located under the following path.

```
c:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\extras\visual_studio_integration
     \MSBuildExtensions\
```

These are the “CUDA 5.0.props” and the “CUDA 5.0.targets”. Also the file “host_config.h” has to be modified, which is located in:

```
c:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include\
```

After that, the modified “props”-file has to be set in the “Build Customizations”-Dialog of the desired project. This loads the customized build steps and executes them, if the project is build. To ensure that all the needed files are visible to the compiler, the last four steps of the point 3 on the NortuRE-Blog [8] have to be applied. To compile the CUDA code, which should only be located in .cu-files, these CUDA-source files should be set as “CUDA C/C++” in the Properties Dialog under:

```
Configuration Properties → General → Item Type
```

To compile a version of OpenCV the last steps should be applied to all modules which uses CUDA functionality. These are the following modules:

- opencv_core
- opencv_gpu

This was done and tried to compile these modules. But it seems that the generated project file creates some internal environment variables with wrong values, which are told to the nvcc. Especially the variable `RelativePath` seems to cause some problems where this variable depends on a correct folder structure. To fix this problem, it should be tried to examine what is wrong with the folder structure in Project 8.

4.1.4 Project Configuration

To use OpenCV in a software project, there are several options to be configured. Create a so called "Property Sheet" to collect all needed configurations in one file, which then can be imported to the project settings.

To create such a Property Sheet, we open the Property Manager in Visual Studio and select the project and the desired build configuration. With Right-Click and "Add New Project Property Sheet…" we can create and add a new Property Sheet. This can now be configured and added to other projects or build configurations. In the case of OpenCV, the following entries are needed:

```
Debug | x64 | OpenCV 2.4.3rc Build Config

Common Properties / VC++ Directories /
Include Directories:      D:\Programme\OpenCV\PreBuild243rc\opencv\build\include
Library Directories:      D:\Programme\OpenCV\PreBuild243rc\opencv\build\x64\vc10\lib

Common Properties / Linker / Input /
Additional Dependencies:  opencv_highgui243d.lib
                          opencv_video243d.lib
                          opencv_ml243d.lib
                          opencv_legacy243d.lib
                          opencv_imgproc243d.lib
                          opencv_objdetect243d.lib
     D:\Programme\OpenCV\PreBuild243rc\opencv\build\gpu\x64\vc10\lib\opencv_core243.lib
     D:\Programme\OpenCV\PreBuild243rc\opencv\build\gpu\x64\vc10\lib\opencv_gpu243.lib
```

In case of the release configuration, we have only to remove the "d" behind the dependencies, which stands for the debug version of the libraries.

Because these libraries do reference to dynamic libraries, the System Environment Variable Path has to be extended by the path to those DLLs like follows:

```
d:\Programme\OpenCV\PreBuild243rc\opencv\build\x64\vc10\bin\
d:\Programme\OpenCV\PreBuild243rc\opencv\build\gpu\x64\vc10\bin
```

In case we want to use the TBB library, we also add the path to these DLLs:

```
d:\Programme\tbb41_20121003oss\bin\intel64\vc11
```

Finally, if no compatible OpenCV version is available yet, there is only to change the toolset settings in each project to Visual Studio 2010.

4.2 Parallelization

This chapter shows tools, frameworks and techniques for parallelization of C++ Code in Visual Studio. There is a little comparison in the end, which shows the Pro and Contra in relation to the requirements.

4.2.1 Concurrency Visualizer

To analyse the produced and parallelized code, a useful tool is the Concurrency Visualizer of Visual Studio 2012. It collects several data of the code execution and then visualizes it in a clearly arranged plot over time. To use it with additional tags, which mark special events in the code, the "Concurrency Visualizer SDK" is used. The MSDN-Blog [9] provides a short introduction to this tool.

The "Concurrency Visualizer SDK" has to be added to the project. This is done at the following menu:

```
Analyze → Concurrency Visualizer → Add SDK to Project …
```

Next step is to include the `cvmarkersobj.h` header and use the namespace `Concurrency::diagnostic`.

There are three ways to tag something in the code:

- Span
- Flag
- Message

Span marks a time-interval with a start and an end. Flag and message on the other hand, mark only a single event at a given time. Moreover, a message can automatically be written to a log-file. To use some of these methods, one has to create an instance of the class `marker_series` in the mentioned namespace, which is needed by all three variants:

```
using Concurrency::diagnostic::marker_series;

// Used by all variants
marker_series series;

// Flag
series.write_flag(_T("some flag-text"));

// Message
series.write_message(_T("some message-text"));

// Span
{
      span s_spanName(series, _T("some span-text"));
      // Code which has to be measured
}
```

To start an analyse-session, the following menu is used:

```
Analyze → Concurrency Visualizer → Start with current Project
```

An example of a CVTrace is shown in Figure 17. The green diamond visualizes a flag, the little grey trapezoid shows the message and the blue block symbolizes the duration of the span.

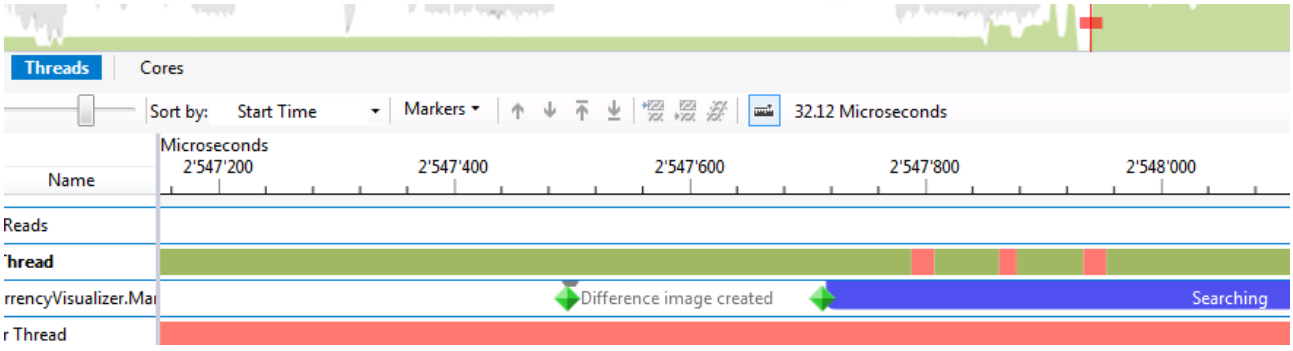


Figure 17 - Example of a CVTrace

For further use and customizations, there is a post [10] on the MSDN-Blog which shows advanced visualization techniques.

The big advantages of the Concurrency Visualizer are the simple use of the basic tools and the diversity of what can be done. But the lead-in to this technology is a little complicated, because of the huge flood of information that is delivered by this tool.

4.2.2 OpenCV

OpenCV [11] is a big library of useful functions for computer vision, and because many picture operations are easy to parallelize, the most of these are already prepared for multicore processors or GPGPU processing.

In fact, there is the namespace `cv::gpu`, which has all pendants of functions that can be used on GPU. Therefore nVidia CUDA is used, so only GPUs of this vendor are supported. Most of the functions have the same name and divers only by the used data class, which is not the `cv::Mat` but the `cv::gpu::GpuMat` class. It is to note, that when using the `GpuMat`, one has to upload the image data to the GPU with the function `GpuMat::upload`. To get the picture back from the GPU to a normal `Mat`, the function `GpuMat::download` is used. An overview of all ported functions can be found at OpenCV Wiki [12].

On the other hand, we can use the power of parallelization without a CUDA GPU. Therefore OpenCV uses the Intel Threading Building Blocks (TBB) Library in many of their functions. To activate TTB, one has only to install TBB, with a minimum version of 2.2 and add the DLL-path to the system variable `PATH`. If installed, OpenCV uses the for TBB optimized functions, where they are available.

OpenCV delivers many simple and also complicated functions of computer vision and related areas. It does a great job at this, but an inexperienced user needs some time to learn all pitfalls. Also the documentation is a bit puzzling at the beginning.

4.2.3 Intel TBB

Intel's Thread Building Blocks (TBB) [13] is a library with a special runtime in the background, which schedules specified jobs and runs them in parallel. To specify these jobs, at most of the time, a `for` loop is converted to a `parallel_for` template call, which takes over the whole job generation and scheduling with the help of the runtime. Besides the used runtime of TBB, it is very similar in its use to OpenMP. The advantages of TBB are that it is open source and already used for parallelization in OpenCV. A little disadvantage is the use of a runtime, which have to be installed on all systems which uses the application.

4.2.4 Microsoft AMP

Microsoft's C++ Accelerated Massive Parallelism (AMP) technology provides a framework for parallelization of code, both on CPU and on GPU. AMP is one big advantage of the new Visual Studio 2012 and uses also some Elements of the new C++ 11 standard.

The AMP framework can be used with the `amp.h` header file and the namespace `concurrency`. A short introduction and advanced thematic, can be found on the MSDN website [14]. To be able to debug GPU code on the delivered software emulator, Windows 8 is required. To debug GPU code on hardware, the driver of the GPU and a DLL with the debugging information has to be installed.

A big advantage of Microsoft AMP is that it uses the GPGPU approach and the more abstract principle of tasks than threads. This lets the user concentrate on the parallelization of the different jobs and he does not need to think about the handling of the threads. A relevant disadvantage is that at the moment it is not possible to compile OpenCV with CUDA and Visual Studio 2012 support, therefore AMP will be used first in Project 8.

4.2.5 OpenMP

OpenMP [15] is a compiler specification for FORTRAN and C/C++, which allows parallelization of sequential code with simple code annotation. These annotations are read by the compiler, which produces parallelized code. The use of OpenMP is very similar to TBB. A handicap is the fact, that OpenMP can only be used with a compiler, which supports it. Many do so, but for example the Visual Studio Compiler supports only until version 2.0 of OpenMP. This is sad, because OpenMP from version 3.0 supports the useful abstraction of threads by tasks, like Microsoft AMP does. Therefore a compatible compiler is always required to use all desired features of the OpenMP standard. Nevertheless the simple use and the fact that no installation is necessary are a big advantage of OpenMP.

4.2.6 nVidia CUDA

The Compute Unified Device Architecture (CUDA) [16] is a framework of nVidia, to program code for graphic adapters with nVidia GPUs. With GPGPU programming not only the CPU, but also the GPU can be used for calculations that have nothing to do with graphics. For this purpose a developer writes a kernel, which describes the algorithm to process over a set of data. The speciality of GPUs is that they are built in a way, that it is possible to use the SIMD paradigm.

Because of the fact, that OpenCV is an image processing library and images can be processed in parallel, it makes sense that OpenCV offers many image processing functions for execution on GPU. Therefore OpenCV uses CUDA, so only nVidia GPUs can be used. But at the time, efforts are done to support OpenCL, which is a unified language for GPU programming. The already stable CUDA functions can be used with the `gpu.h` header and the `cv::gpu` namespace.

CUDA is a powerful tool to use all resources of a modern PC. But it is not simple to dive into GPU computing. Therefore it is nice to use the power of CUDA by using the corresponding functions of OpenCV.

4.2.7 AMD APP

AMD's Accelerated Parallel Processing (APP) [17] is a technology to unify the CPU and GPU. In fact it uses OpenCL as language to do GPGPU programming on AMD hardware. Additionally Microsoft AMP or AMD's own C++ template library Bolt [18] can be used for using APP. Below APP sits the Heterogeneous System Architecture (HSA) of AMD [19] [20], which is a heterogeneous computer system, which combines the power of the CPU and GPU. As nVidia's CUDA, AMD's APP also only works on his own GPUs. Because this technology has been used very little in our institute and much more experience with nVidia CUDA is available, no use of it is considered. But it could be very interesting to dive into this technology, especially because it uses the principle of unified GPGPU.

4.2.8 TinyThread++

TinyThread++ [21] is a portable open source C++ library, which implements threads conform to the C++11 standard but can compiled also with non C++11 compatible compilers. It is a nice little library with only the essentials to use threads and mutexes. Source code that is already written with the use of TinyThread++ can easily be converted to C++11 code with the corresponding `std::` classes.

4.2.9 Comparison

All the described technologies have its good and bad sites. With all these mentioned, OpenCV has been chosen because it is a big and fully featured library of the needed functionality, which in addition delivers some CPU parallelization and GPU acceleration where they are useful. For that, TBB and nVidia's CUDA are used. In addition, the already known OpenMP and the new TinyThread++ is used to do some of the own parallelization. Microsoft AMP will be used in Project 8, when there hopefully will be a solution for the compilation problem. And at the end, AMD's APP seems to be a very interesting new approach for GPGPU programming, and could be useful for other new projects, which do not need OpenCV.

4.3 Speed Optimization

Person recognition in real time video depends heavily on a good performance and the entire use of all available resources. Several approaches and different technologies have been used to reach this goal. This chapter shows important performance optimizations and how they have been implemented. Problems which could not been solved yet or are not relevant for this project are also mentioned.

4.3.1 Initialisation

This point has nothing to do with video processing or person recognition at all. It is a general considered fact that all initialization of parts which do not change again and are needed later for fast processing, should be prepared on the program start-up. In another case, a function should do the initialisation process only at the first call or later, when the conditions have changed. Such could be not obvious and need analysis of already written code to optimize it.

The initialization parts in this application are the following:

- Initialize CUDA environment
- Open video file / stream
- Create GUI window
- Calculate chunks of the image

Especially the first point could be important for measuring the performance of CUDA powered functions. That is because the first call of a CUDA functions initializes the environment of the GPU and thus can take much time. Figure 18 shows the first seconds of the application execution. The CUDA start-up can be evaluated to duration of nearly two seconds.

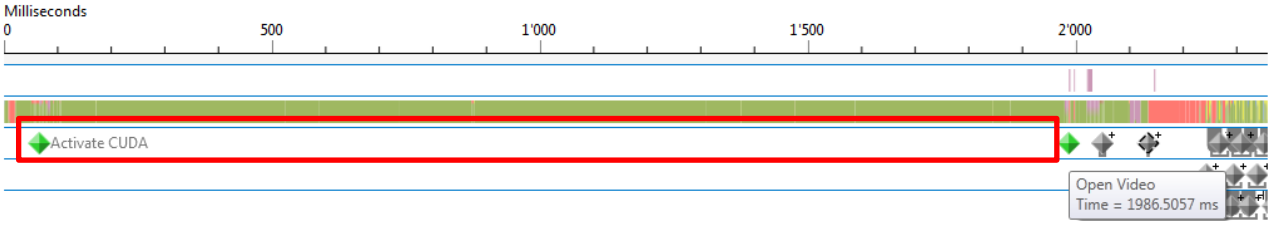


Figure 18 - Initialization of Application

To prepare the environment before the GPU is actually used, get the CUDA enabled device count (`cv::gpu::getCudaEnabledDeviceCount`), or check the device status (`cv::gpu::getDevice`) for instance. This ensures that the first productive CUDA function will have the full available performance.

4.3.2 Video Buffer

The video processing, that is done in this application, sometimes needs more than the current image of the stream. It would also be useful to buffer the next frames, in order that they are ready if needed. Another advantage of buffering is to do the pre-processing of each frame meanwhile. The written class `people::VideoBuffer` delivers this functionality. Figure 19 shows the structure of this class. In principle, it stores a defined number of frames in its local storage. Some cells of this memory are used for previous images and some for the next ones. The number of this container can be defined with the private constants: `mPrevBufferSize` and `mNextBufferSize`. Additional, there is always one cell for the image that is currently processed. All this stored frames are collected from the defined file, converted to grey and resized. They are further saved to each type of memory: CPU and GPU. This is very useful if the images are later needed for different processing steps, which sometimes can only run on CPU.

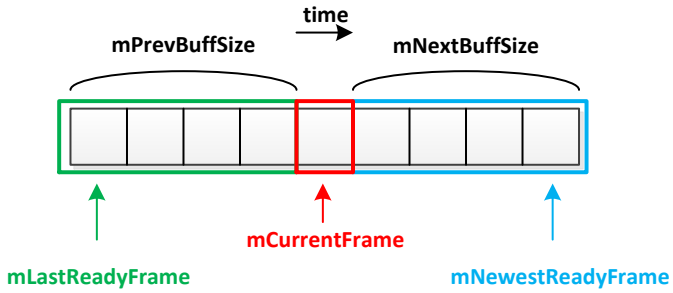


Figure 19 - Structure of the VideoBuffer

Because the buffering and pre-processing is always the same, it can easily be optimized. First of all, the GPU has been used to load the frame from the file. After that, all pre-processing steps are done on GPU and the finished frames are downloaded to the CPU memory at last. The measuring in Figure 20 shows this process, which needs about 3 milliseconds. The class `cv::gpu::VideoReader_GPU` is used to load the frames directly to the GPU memory.

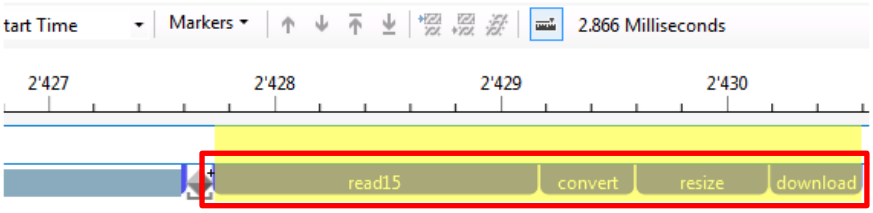


Figure 20 - Duration of Buffering with VideoReader_GPU

Another tested variant was to load the frames first to the CPU and upload it then to GPU memory. The pre-processing steps could then be made on GPU or on CPU, depending on which version is faster. This is showed in Figure 21 with the class `cv::VideoCapture`. All pre-processing steps are done on the GPU because these functions are always faster thereon. The whole buffering with this approach is slower and uses between 5 and 7 milliseconds.

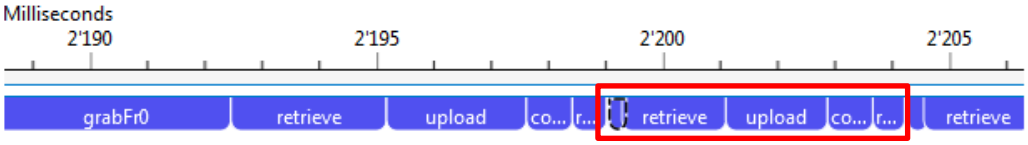


Figure 21 - Duration of Buffering with VideoCapture

Besides the optimization of the speed itself, the parallelization can bring some improvements. Because of the fact that the video file can only be accessed sequentially and the whole buffering should not affect the actual person detection, it is useful to create a specialized thread which does the buffering parallel to all other operations. The TinyThread++ library is used for this purpose. The buffer thread is concerned for the buffering and sleeps, if the memory of the buffer is full, no more frames are in the video stream or the affected memory cells are accessed by the main process. Because the buffer is empty at the start of the program, it has to read many frames at once. This start-up sequence, after the call of `startBuffering`, is shown in Figure 22. In this example, the video is configured to start at frame 10. The buffer has both sizes set to 4 and therefore needs to buffer frame 6 first (yellow bars). The previous frames are flushed (blue bar). The main application wants to process the frame 10 and waits for this (cyan bars). After the buffering of frame 10, the main application starts to process it and the buffer continues with the buffering of the next frames in parallel.

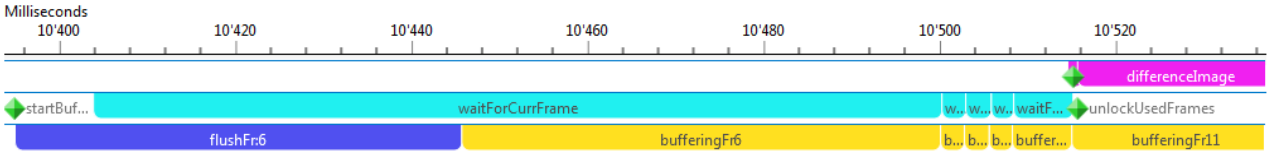


Figure 22 - Start of Buffering

After the buffer is once filled, the processing and the corresponding performance go into a stabilized order. Figure 23 shows such iteration with motion detection, displaying and buffering of an image. In addition, the time of a typical buffer step (yellow bar) can be evaluated with about 3.1 milliseconds.

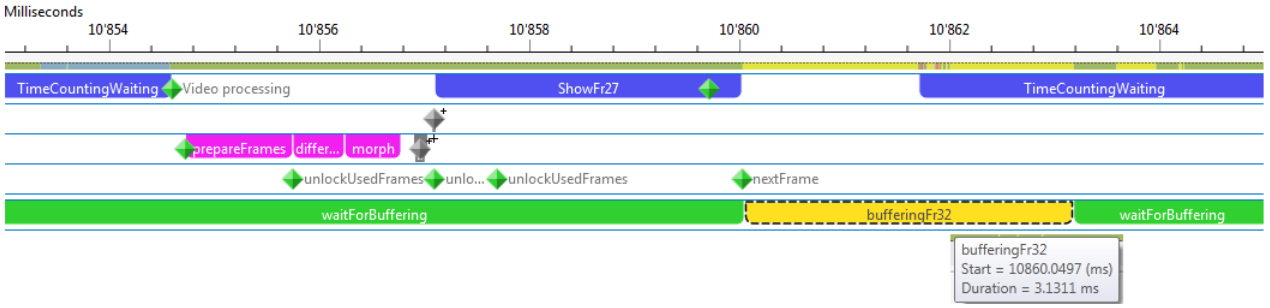


Figure 23 - Processing of one frame

4.3.3 GPU Stream

The previous shown way to load the CPU and GPU can be taken even further with the class `gpu::stream`. This class lets the user queue up many steps of GPU processes. For instance, the next four frames and their pre-processing steps could be assigned asynchronous to the GPU. In this time the CPU could do other things until the GPU has finished these tasks. Unfortunately this is not possible at the moment, because the only prepared functions for that are the following:

- `Download`
- `Upload`
- `Copy`
- `MemSet`
- `Convert`

Out of these, the `upload` and `download` function only can handle tasks with different input data. Therefore even the used function `convert` cannot be used because it is unable to handle more than one image at the time. Finally, the only improvement that could be achieved, is the asynchronous call of the `convert` function which is only a small task of 0.4 milliseconds that can be observed in Figure 20.

4.3.4 Irrelevant Performance Issues

This paragraph only mentions some time expensive processes that are used in the application, but which would have only small influences in a real application. These are the functions `cv::imshow` and `cv::waitkey`. These two calls are needed to show the image on the screen. Because the system behind the `namedWindow` and `imshow` needs some processing time to update the screen, `waitkey` has to be called after each `imshow`, to let the display be updated.

These parts can be ignored because a real application would not handle the displaying of the processed images but only deliver the positions and names of the recognized people in the video.

5 Discussion

5.1 Task

The task of this master project is to determine, if a video stream can be processed in real time and recognize the people that show up in there, only with the resources of a personal computer. In the first part out of three, the main goal was to gain knowledge of the available technologies and how to use them. The desired purposes of these were: parallelization, computer vision and GPU computing.

Besides that, a video collection had to be recorded with some different persons, locations and situations. This collection is needed to develop and test the application.

With this groundwork, software had to be developed which are able to detect a person and the direction of its head in the video. This is then used in the later parts to build a head model and to recognize the people in the video.

The detailed task definition can be observed in Appendix A.

5.2 Solutions & Results

To gain knowledge of all the available technologies, several investigations had been done. A big part was the use of these technologies, to gain experiences. Besides the Concurrency Visualizer, technologies with already existing know-how were chosen. Also several libraries, especially OpenCV, had dependencies to others, so they had to be taken too.

The test data collection was a small task but needed to be well planned. In the end, not all desired situations could be recorded, but the most part which should be enough, until the software is no longer a student project and becomes a real application.

The biggest part of this project was the person detection. This had begun with a simple motion detection which was iteratively improved and is now complete. But the detection of the head and his direction could not be solved. Only a few ideas and concepts had been developed. But with these, a fast start in the Project 8 should be possible.

A reason for this was that much time was spent for the compilation problems of OpenCV with CUDA 5 and the Visual Studio 2012 compiler. In addition, the familiarization with OpenCV was not so easy then thought at the beginning.

5.3 Future work

As already mentioned, the detection of the head and its direction will be the next step in Project 8. After that, a head model and a database to store all these head data needs to be developed. This step will probably need much testing to improve the detection rate and speed.

Also the missing use of Microsoft AMP with the new compiler and a compatible OpenCV compilation should be realised in the next project.

Index

List of references

[1] OpenCV, “OpenCV Docs: Clustering,” [Online]. Available: http://docs.opencv.org/modules/core/doc/clustering.html?highlight=partition. [Accessed 26. January 2013].

[2] Wikipedia, “Sweep Line Algorithm,” [Online]. Available: http://en.wikipedia.org/wiki/Sweep_line_algorithm. [Accessed 26. January 2013].

[3] Z. Kalal, “TLD Vision,” [Online]. Available: http://tldvision.com/. [Accessed 26. January 2013].

[4] Z. Kalal, “YouTube: Predator: Camera That Learns,” [Online]. Available: http://www.youtube.com/watch?v=1GhNXHCQGsM. [Accessed 26. January 2013].

[5] Z. Kalal, “Github OpenTLD,” [Online]. Available: https://github.com/zk00006/OpenTLD. [Accessed 26. January 2013].

[6] OpenCV, “Installation of OpenCV in Windows,” [Online]. Available: http://docs.opencv.org/trunk/doc/tutorials/introduction/windows_install/windows_install.html. [Accessed 26. January 2013].

[7] Coding Gorilla, “Developing CUDA and C++ AMP in Visual Studio 2011,” [Online]. Available: http://codinggorilla.domemtech.com/?p=1112. [Accessed 26. January 2013].

[8] NortuRE Blog, “GPU Parallel Programming in VS2012 with NVIDIA CUDA,” [Online]. Available: blog.norture.com/2012/10/gpu-parallel-programming-in-vs2012-with-nvidia-cuda/. [Accessed 26. January 2013].

[9] J. Rapp, “Introducing the Concurrency Visualizer SDK,” Microsoft, [Online]. Available: http://blogs.msdn.com/b/visualizeparallel/archive/2011/10/17/introducing-the-concurrency-visualizer-sdk.aspx. [Accessed 17. Dezember 2012].

[10] J. Rapp, “Concurrency Visualizer SDK: Advanced Visualization Techniques,” [Online]. Available: http://blogs.msdn.com/b/visualizeparallel/archive/2011/11/03/concurrency-visualizer-sdk-advanced-visualization-techniques.aspx. [Accessed 15. Januar 2013].

[11] OpenCV, “OpenCV Wiki,” [Online]. Available: http://code.opencv.org/projects/opencv/wiki. [Accessed 26. January 2013].

[12] OpenCV, “OpenCV_GPU,” [Online]. Available: http://opencv.willowgarage.com/wiki/OpenCV_GPU. [Accessed 26. January 2013].

[13] Intel, “Threading Building Blocks,” [Online]. Available: http://threadingbuildingblocks.org/. [Accessed 26. January 2013].

[14] MSDN, “C++ AMP Overview,” Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/hh265136.aspx. [Accessed 26. January 2013].

[15] OpenMP, “OpenMP,” [Online]. Available: http://openmp.org/wp/. [Accessed 26. January 2013].

[16] nVidia, “CUDA Toolkit,” [Online]. Available: https://developer.nvidia.com/cuda-toolkit. [Accessed 26. January 2013].

[17] AMD, “Accelerated Parallel Processing (APP) SDK,” [Online]. Available: http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/. [Accessed 26. January 2013].

[18] AMD, “Bolt C++ Template Library,” [Online]. Available: http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/bolt-c-template-library/. [Accessed 26. January 2013].

[19] AMD, “HSA Foundation,” [Online]. Available: http://developer.amd.com/community/hsa-foundation/. [Accessed 26. January 2013].

[20] AMD, “HSA Developer Tools,” [Online]. Available: http://hsafoundation.com/hsa-developer-tools/. [Accessed 26.

January 2013].

[21] Marcus, “TinyThread++,” [Online]. Available: http://tinythreadpp.bitsnbites.eu/. [Accessed 26. January 2013].

[22] C. Lang, “Vorgehen_TestdatenErzeugen_Satz1.xlsx,” 2012.

[23] D. Baur, “Automatische Gesichtserkennung: Methoden und Anwendungen,” Universität München, Amalienstrasse 17, 80333 München Deutschland.

[24] A. Venkatraman, “enVisage : Face Recognition in Videos,” Imperial College, University of London, 2006.

[25] E. Murphy-Chutorian and M. M. Trivedi, “Head Pose Estimation in Computer Vision: A Survey,” IEEE Transactions on Pattern Analysis and Machine Intelligence, 2008.

[26] N. M. S. and S. D. V., Pattern Recognition: An Algorithmic Approach, London: Springer, 2011.

[27] P. Salembier and T. Sikora, Introduction to MPEG-7: Multimedia Content Description Interface, New York, NY, USA: John Wiley & Sons, Inc., 2001.

[28] J. Lee, Automatic Video Management System Using Face Recognition and MPEG-7 Visual Descriptors, ETRI Journal 27, 806-809, 2005.

[29] M. P. Da Silva, V. Courboulay, A. Prigent and P. Estraillier, “Fast, low resource, head detection and tracking for interactive applications,” PsychNology Journal, Université de La Rochelle, 2009, Vol.7, Nr. 3.

[30] P. N. Belhumeur, J. P. Hespanha and D. J. Kriegman, “Eigenfaces vs. Fisherfaces: Recognition using class specific linear Projection,” IEEE Trans. , Yale University, New Haven, 1997.

[31] T. T. Will, C++ 11 programmieren, Bonn: Galileo Press, 2012.

[32] R. Laganière, OpenCV 2, Computer Vision Application Programming Cookbock, Birmingham: Packt Publishing Ltd., 2011.

Declaration

I confirm that I have made this work independently. Information that is taken verbatim or according to other sources has been specified.

Place, Date

Windisch, 27.01.2013

Signature

Lang Christian

Appendix

A. Project Clarification

This appendix contains the original German project clarification which was created at the beginning of the master project. It describes the motivation, the goal and some proposals of proceeding.

1. Ausgangslage

Im Rahmen des gesamten Masterstudiums sollen Erfahrungen mit Parallelisierung und Auslagerung von Algorithmen für die Personenerkennung auf GPU gemacht werden. Dazu sollen Technologien zur massiven Parallelisierung von Computer Vision Aufgaben evaluiert und zudem Algorithmen und Vorgehen für die Personenerkennung in Videos gesucht, implementiert und parallelisiert werden.

2. Aufgabenstellung

Im ersten Teil dieser Arbeit geht es um eine Einarbeitung in die zu verwendenden Technologien (CUDA, OpenCL, AMP) und ins Themengebiet der situationsunabhängigen Personenerkennung. Diesbezüglich interessiert vor allem, wie Personendaten aus Videos extrahiert und modelliert werden sollen, so dass sie zu einem späteren Zeitpunkt für eine situationsunabhängige Personenerkennung verwendet werden können.

3. Organisation

Auftraggeber**:** Prof. Dr. Christoph Stamm

Institut für Mobile und Verteilte Systeme, FHNW

Student**:** Christian Lang

christian.lang1@students.fhnw.ch

4. Ziele

Folgende Ziele sollen erreicht werden:

1. Liste der verfügbaren Technologien für GPU-Programmierung und dessen Vor- und Nachteile erstellen. Auswahl der Technologie für die GPU-Programmierung.
2. Erstellen einer Testdatensammlung bestehend aus Videos mit unterschiedlichen Personen, Perspektiven, Bewegungen, Gesichtsbedeckungen, Lichtverhältnissen.
3. Auswahl eines geeigneten Algorithmus zur Detektion eines Kopfes und dessen Blickrichtung. Parallele Implementation dieses Algorithmus, so dass in einem Live-Video der Kopf markiert und der Aufnahmewinkel relativ zum Blickwinkel angegeben wird. Z.B. Rahmen um Kopf mit Bemerkung „+90°“.

5. Teilaufgaben

Der gesamte Projektumfang beträgt 14 ECTS und somit 420 Arbeitsstunden. Die Arbeit gliedert sich in die folgenden Unterkapitel:

a. Wissensaufbau Parallelisierung & GPU

Da die Problematik zur Personenerkennung komplex und deshalb rechenintensiv ist, wird neben einem effizienten Algorithmus auch eine effektive Ausnutzung der Hardware vorausgesetzt. Hier möchte man vor allem die in beinahe jedem PC enthaltene Grafikkarte (GPU) geschickt einsetzten. Diese ermöglicht ein massives paralleles Rechnen und ebnet so den Weg zur schnellen Personenerkennung in Videos. Um die leistungsfähige Hardware sinnvoll und effektiv verwenden zu können, muss sich in die Architektur und Funktionsweise von GPUs eingearbeitet werden. Diese arbeiten nach dem Prinzip „Single Instruction, Multiple Data“ (SIMD) und bieten dadurch einen erheblichen Vorteil für unsere Aufgabe, da sie eine Funktion parallel auf mehrere Datensätze anwenden kann, was bei der Bildverarbeitung häufig der Fall ist. Um die einzelnen Aufgaben sinnvoll auf die Recheneinheiten des PCs zu verteilen, soll auch das Prinzip von „General Purpose Computation on GPU“ (GPGPU) miteinbezogen werden.

b. Evaluation einer Technologie für Parallelisierung mit GPU-Unterstützung

In Bezug auf die Anwendung von Parallelisierung und Berechnungen auf einer GPU, sollen folgende Technologien und Bibliotheken analysiert und erlernt werden:

- nVidia‘s proprietäre Sprache CUDA
- AMD‘s proprietäre Sprache APP (AMD Accelerated Parallel Processing)
- Standard OpenCL
- Microsoft Bibliothek C++ AMP (Accelerated Massive Parallelism)

Dies sind die aktuellsten Technologien zur Programmierung auf GPUs. Zudem bieten sie teilweise eine starke Abstraktion aller Recheneinheiten eines PCs, wodurch die Komplexität der Programmierung gesenkt werden kann. Jedoch ist neben der Anwenderfreundlichkeit vor allem die resultierende Leistungsfähigkeit für die Anwendung in Videos wichtig, weshalb durch Sammeln solcher Performancedaten eine Bewertung aller Technologien ermöglicht werden soll.

Diese Bewertung wird benötigt, um die geeignetste Technologie für unseren Anwendungsfall auszuwählen.

c. Testdaten erzeugen

Der zu entwickelnde Algorithmus zur Personenerkennung muss mit entsprechendem Material überprüft werden. Dazu soll ein Testsatz an Videos mit unterschiedlichen Personen und Rahmenbedingungen aufgezeichnet werden. Es sollen folgende Fakten variiert werden:

- Personen (Geschlecht, Haarfarbe, Hautfarbe, Grösse)
- Anzahl Personen im Bild (keine, eine, mehrere)
- Laufwege der Personen (Kreuzen von mehreren Personen, Distanz zur Kamera)
- Kopfhaltung und -bewegung der Personen
- Brille, Hut, Schal, etc.
- Aufnahmewinkel der Kamera

Die genaue Planung erfolgt in einem separatem Dokument [22]. Zusätzlich sollen alle Aufnahmen im Nachhinein unterschiedlich konvertiert werden, um weitere Variationen zu erhalten:

- Farbig, Schwarz-Weiss
- Auflösung reduzieren
- Evtl. Bildwiederholungsrate

Das aufgenommene Videomaterial soll danach so aufgeteilt werden, dass der eine Teil als Trainingsmenge, der andere als Testmenge verwendet werden kann. Dazu müssen beide Teile eine gleichverteilte Anzahl von alle unterschiedlichen Szenen enthalten. Welche Personen darauf erscheinen sollte ebenfalls

ausgeglichen sein. Somit kann mit der Trainingsmenge z.B. eine Person in einer bestimmten Situation „gelernt“ und die gleiche Person dann in einer anderen Situation in der Testmenge erkannt werden. So kann ausgeschlossen werden, dass eine Person nur in der „gelernten“ Umgebung korrekt erkannt wird.

d. Kopfdetektion und Modellgenerierung in Videos

Videos haben einen erheblich grösseren Informationsgehalt als Bilder. Denn anders als ein Bild, ist ein Video eine Reihe von Bildern, welche ein Objekt nicht nur aus einer, sondern aus mehreren Perspektiven und mit unterschiedlicher Skalierung und Beleuchtung zeigt. Dieses Potential soll zukünftig für die situationsunabhängige Personenerkennung verwendet werden können. Der durch die grosse Informationsmenge erhöhte Rechenaufwand, um die Informationen zu verarbeiten, soll mit der Parallelisierung und GPU-Unterstützung bewältigt werden.

Bezogen auf die Erkennung von Personen sind bereits sehr viele Erfahrungen mit Gesichtserkennung gemacht worden. Eine Übersicht stellt die Arbeit von Baur [23] vor und die Arbeit von Venkatraman [24] beschäftigt sich mit der Erkennung in Videos. Unterstützend dazu existiert OpenCV, eine von Intel initiierte OpenSource Bibliothek, welches sehr viele Vorgehensweisen bereits implementiert. Unter anderem sind Gesichtslokalisierung und -erkennung bereits verwendbar und mithilfe der CUDA-API auch auf der GPU ausführbar. Die auf der GPU ausführbaren Funktionen von OpenCV sind auf dessen Homepage [12] beschrieben. Eine Alternative zur OpenCV-Bibliothek wäre das Accord.NET Framework, welches das C# Framework AForge.NET um Gesichtserkennung etc. erweitert.

Anders als ein Bild, bietet ein Video jedoch nicht nur ein statisches Abbild eines Gesichts, sondern auch Bewegungen und Stimme und zudem unterschiedliche Ansichten des Gesichts einer Person. Deshalb soll sich hier nicht nur auf ein Bild beschränkt werden, sondern ein Modell erstellt werden, welches möglichst viele Merkmale der Person aus den unterschiedlichen Ansichten des Kopfes im Video extrahiert.

Dazu muss in einer ersten Phase der Kopf detektiert und verfolgt werden. Diverse Vorgehensweisen werden von Murphy [25] vorgestellt. Alle Bilder, auf welchen der gleiche Kopf sichtbar ist, sollen in einer zweiten Phase verwendet werden, um die markanten Merkmale des Kopfes zu extrahieren. Welche Daten relevant sind und wie sie verarbeitet werden sollen, wird in einem Datenmodell des Kopfes festgelegt. Dieses Datenmodell soll dann im Folgeprojekt die Grundlage für die Personenerkennung bilden. Der grobe Ablauf in der vollständigen Software ist in Abbildung 1 gezeigt.

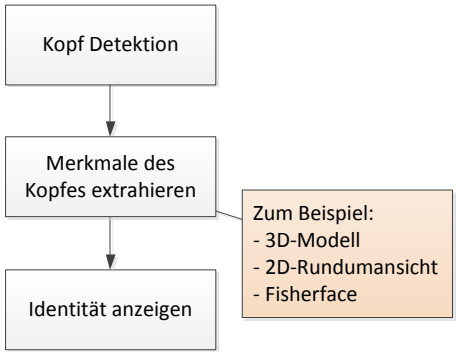


Abbildung 1 - Ablauf der Identifikation einer Person im Video

Um im letzten Schritt die im zu analysierenden Video erfassten Daten zur Personenerkennung verwenden zu können, wird ein System benötigt, welches die Kopfdaten einer nicht identifizierten Person einem bekannten Datensatz zuweist. Auch hier existieren unterschiedliche Vorgehensweisen welche im Buch von Narasimha Murty et al [26] vorgestellt werden. Dieses System soll entscheiden, ob die zu vergleichenden Kopfdaten zu einem abgespeicherten Datensatz passen und somit, wer auf dem aktuellen Video zu sehen ist.

Die Schwierigkeit dieses Projekts ist einerseits den Kopf zu detektieren, andererseits ein geeignetes Datenmodel zu erstellen, welches bestimmt, was die markanten Merkmale eines Kopfes sind und dabei

aber nicht allzu ressourcenverschwenderisch ist. Eine sehr vereinfachte Variante wäre es, jeweils ein Bild des Kopfes aus unterschiedlichen Winkeln zu speichern, wie dies in Abbildung 2 gezeigt wird.

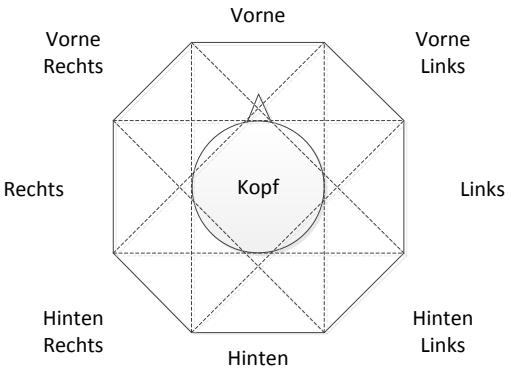


Abbildung 2 - Entwurf eines Kopfmodells, welches max. 8 Bilder speichert

Hinweise auf ein geeignetes Modell können aus anderen Anwendungen wie z.B. in MPEG-7 [27] [28] oder allgemeinen Algorithmen zur Gesichtserkennung [23] [29] wie z.B. Fisherfaces [30] herausgearbeitet werden.

Wichtig ist, dass das Datenmodell nicht zwingend alle Daten gleichzeitig für das Erstellen eines Datensatzes benötigt, da in einem Livesystem am Anfang nur ein einziges Bild von einem Kopf verfügbar ist. Zudem weis man nicht, aus welchem Winkel dieses erste Bild ist, aber auch nicht aus welchem Winkel die folgenden Bilder sein werden. Somit soll zu jedem Zeitpunkt ein Datensatz erstellt und auch mit neuen Informationen ergänzt werden können. Dieser Vorgang wird in Abbildung 3 symbolisiert.

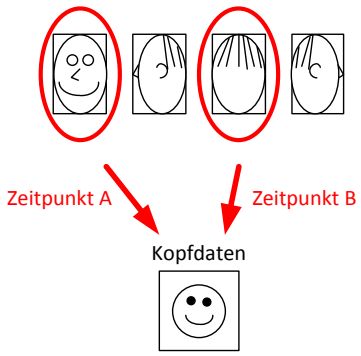


Abbildung 3 - Erstellen eines Kopf-Datensatzes

e. Andere und weiterführende Themen

Zu beachtende Techniken und Theorien welche die Aufgabe der Personenerkennung unterstützen könnten sind Folgende:

- Pattern Recognition
- Pulsbasierte neuronale Netze
- EBGM (Elastische Bündelgraphenmatching)
- Computer Vision Bibliothek OpenCV

f. Ausblick

Im Projekt 8 soll, aufbauend auf der Detektion des Kopfes, eine geeignete Variante gefunden werden, wie Erkennungsmerkmale extrahiert und in einem Datensatz des Kopfes gespeichert werden kann. Im P9 soll dann ein Pattern-Matching-Algorithmus gewählt und implementiert werden, welcher die Merkmale einer

Person zuweist und somit die Identität des Kopfes im Bild bestimmt. Die dabei entstehenden Implementationen sollen aus Performancegründen mithilfe der gewählten Parallelisierungstechnik optimiert werden. Eine Übersicht über alle drei Teilprojekte zeigt die Abbildung 4.

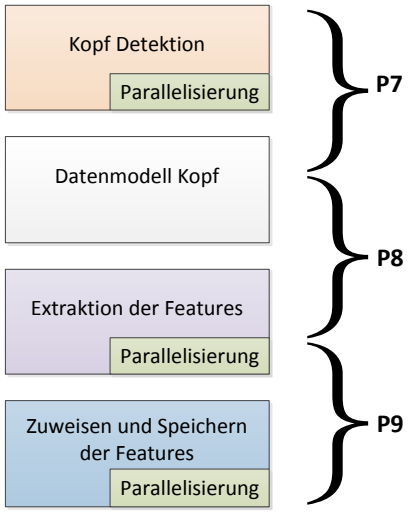


Abbildung 4 - Ablauf Gesamtes Masterprojekt

6. Produkt

Die Dokumentation soll in englischer Sprache verfasst werden und vor allem die Auswahl der Parallelisierungsstrategie für das Projekt 8 rechtfertigen und den Algorithmus für die Detektion des Kopfes und dessen Modellextraktion liefern.

7. Semesterplanung

Das Semester ist folgendermassen verplant:

Semesterwoche		1	2	3	4	5	6	7	8	9	10	11	12	13	14	Ferien				MSP		
Kalenderwoche		38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	1	2	3	4	5	6
MS		16	16	16	16	16	16	16	16	16	16	16	16	16	16	0	24	40	32	24	0	0
Thema	**MS**																					
Klärung	**32**	16	4	4	8																	
Parallelisierung	**92**																					
Allgemeine Theorie	12		8	4																		
Technologien	60		4	8				8	8	8	8						8	8				
Bewertung Techn.	20										8	4							8			
Personenerkennung	**140**																					
Testdaten sammeln	8				4	4																
Kopfdetektion	76				4	12	16	8	8	8		4	8	8								
Kopfmodell	56													8	16		16	16				
Andere CV Themen	**16**																	16				
Dokumentation	**64**											8	8						24	24		

Klärung
Zwischenpräs.
Projektabgabe

Abbildung 5 - Projektplanung

B. Test Data

8. Descriptions of scenes

The following table was used to plan the recorded scenes.

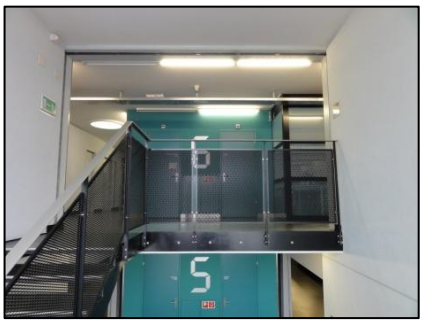
Anzahl Personen	Person 1	Kleidung Person 1	Gesicht Person 1	Person 2..n	Personen	Kamera
Ohne Person						statisch
						rein/rauszoomen
						drehen um vertikale Achse
						drehen um horizontale Achse
Mit 1 Person	steht stil mit Gesicht zur Kamera	ohne Gesichtsbekleidung	Neutral		männlich ohne Bart	statisch
	steht stil mit abgewantem Gesicht	Brille	Lachen		männlich mit Bart	rein/rauszoomen
	dreht sich um eigene vertikale Achse	Sonnenbrille	geschlossene Augen		männlich dunkelhäutig	drehen um vertikale Achse
	läuft einen Kreis vor Kamera	Schal	offener Mund		weiblich lange Haare	drehen um horizontale Achse
	läuft orthogonal auf Kamera zu	Mütze	Blick nach oben			
	läuft orthogonal von Kamera weg		Blick nach unten			
	läuft parallel an Kamera vorbei					
	läuft in Zick-Zack auf Kamera zu					
	läuft in Zick-Zack von Kamera weg					
Mit 2 Personen	steht stil mit Gesicht zur Kamera	ohne Gesichtsbekleidung	Neutral	kreuzt hinter Person 1	männlich ohne Bart	statisch
	steht stil mit abgewantem Gesicht	Brille	Lachen	kreuzt vor Person 1	männlich mit Bart	
	dreht sich um eigene vertikale Achse	Sonnenbrille	geschlossene Augen	läuft orthogonal zur Kamera an Person 1 vorbei	männlich dunkelhäutig	
	läuft einen Kreis vor Kamera	Schal	offener Mund		weiblich lange Haare	
	läuft orthogonal auf Kamera zu	Mütze	Blick nach oben			
	läuft orthogonal von Kamera weg		Blick nach unten			
	läuft parallel an Kamera vorbei					
	läuft in Zick-Zack auf Kamera zu					
	läuft in Zick-Zack von Kamera weg					
Mit n Personen	steht stil mit Gesicht zur Kamera	ohne Gesichtsbekleidung	Neutral /Frei wählbar	bewegen sich frei durch Bild	männlich ohne Bart	statisch
	steht stil mit abgewantem Gesicht				männlich mit Bart	
	dreht sich um eigene vertikale Achse				männlich dunkelhäutig	
	läuft einen Kreis vor Kamera				weiblich lange Haare	
	läuft orthogonal auf Kamera zu					
	läuft orthogonal von Kamera weg					
	läuft parallel an Kamera vorbei					
	läuft in Zick-Zack auf Kamera zu					
	läuft in Zick-Zack von Kamera weg					

9. Descriptions of places

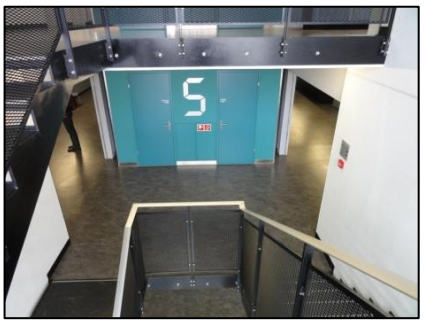
Four places were chosen to use for recording of the test videos. All these places are in or around the building Nord in Windisch.

Flur, Winkel horizontal

Treppenhaus, Winkel nach oberem Geschoss



Treppenhaus, Winkel nach unterem Geschoss



Vor Gebäude Nord, Winkel horizontal