UNIVERSITY OF OSLO
Department of Informatics

Silhouette Extraction using Graphics Processing Units

Master's Thesis

Kristian Haga Karstensen

Contents

List of Figures

List of Tables

Listings

Acknowledgements

I would like to thank my advisors Alexander Eichhorn and Pål Halvorsen, for their valuable feedback and guidance during the work with this thesis.

Thanks to Tor Ivar Johansen, Christian Tryti and the rest of the students at the Simula lab for a productive and fun environment.

I would also like to thank my parents for their good support throughout my studies.

A special thank you to my wife Marte for her constant support, encouragement and patience.

Oslo, May 2012
Kristian Haga Karstensen

Chapter 1

Introduction

1.1 Background

A silhouette gives an outline of a person or an object. When extracting silhouettes from an image or video, we acquire information about where in the image the foreground objects are located, their sizes and their shapes. This silhouette information can be useful in many ways.

In computer vision, silhouette extraction plays an important role. Many applications need to extract people or objects, e.g. in video surveillance, tracking of objects, human or object detection [1], 3D-reconstruction [2] or mixed reality [3] applications. For instance in a mixed reality scenario, we may want to reinsert a person into a different setting or location to interact with other people in real-time.

Many of the aforementioned applications use, at least in part, silhouettes (from one or multiple viewpoints) as input. Since the extraction of silhouettes is often a small part of these applications, it has to be done as fast as possible — at least in real-time applications, leaving valuable execution time to the rest of the application.

General-purpose computing on graphics processing units (GPGPU) has become increasingly popular the last years. With development frameworks like CUDA [4] and OpenCL [5], the parallel processing power of GPUs have never been easier to utilize. This gives us the possibility to develop real-time applications using high resolution video.

1.2 Problem Statement

As we will see in chapter 2, many different approaches for extracting and refining silhouettes in different environments have been proposed. Since silhouette extraction is often only one of the needed processing steps in applications where silhouettes are used, high speed silhouette

extraction is needed. Many proposed methods on CPU work well and in real time on low resolution images, and are therefore well suited in applications where low resolution images are adequate.

Many modern applications will also require silhouettes from higher resolution images, and with high frame rates, for example to create 3D models for use in mixed reality scenarios. We therefore want to implement an already existing algorithm for silhouette extraction on GPU, to see how much we can improve the performance compared to a CPU-implementation, aiming at real time execution.

We will perform an experimental approach on our GPU implementation, with different optimizations on the different parts of the algorithm, and present the effects they have on the execution time.

We will also do a simple quality assessment of our algorithm looking at the percentage of correct/incorrect detection of foreground pixels, to see how accurate our chosen algorithm is extracting silhouettes.

1.3 Limitations

Our implementations will be tested on video of sizes 1024x576 and 512x288 pixels, with 25 fps. As the algorithm we have chosen to implement includes many different steps, we will omit the last filtering and refinement steps, to limit our scope a bit.

Our main focus will be on developing a fast algorithm, so performance is our primary goal. Improving the existing algorithm to get higher quality silhouettes is therefore not the main goal of the thesis, even though it is desirable. We discuss some possible improvements in chapter 4 and 8.

1.4 Main Contributions

The main contributions in this thesis are as follows: We explore a silhouette extraction algorithm and implement most parts of this algorithm on both CPU and GPU, with the GPU implementation being our main focus. In our GPU-implementation, we investigate different optimization approaches on each step of the algorithm, measure runtime on different video data and report our results. We show that when implementing image processing algorithms on GPU, multiple implementations should be tried to get the best runtime, as small changes can make pretty big differences.

We show that our GPU implementation is between 2 and 3.4 times faster than our CPU

implementation in the worst case and that we can get around 6 frames per second on video data with size 1024x768. On video with size 512x288 we get a framerate of around 26 fps.

We perform a simple quality assessment of experimental resulting silhouettes of the algorithm, and point at possible optimizations that can be performed to get better results.

1.5 Outline

In chapter 2, we present our definition of silhouette extraction and look at previous work done in this field, presenting different approaches. We then move on to chapter 3, where we take a look at GPGPU and present the OpenCL framework for programming on heterogeneous architectures, focusing on GPU programming. We present the different memory types, execution model, and look at different optimization strategies for GPU-implementation. In chapter 4, we present the silhouette extraction algorithm we will implement on GPU, explaining the different steps of the algorithm in detail. We introduce some optimizations to the algorithm, and look briefly at possible optimization strategies on GPU for the different steps. Chapter 5 presents our GPU implementation, including different optimizations for each step. We also show performance measurements of the different optimizations, and present the total runtime of the algorithm on different video data. Our CPU implementation is briefly presented in chapter 6, and a performance comparison between the CPU and GPU implementation is shown. We show the runtime differences in each of the steps, and finally present the total runtime difference between CPU and GPU versions on different video data. A simple quality assessment of the extracted silhouettes is presented in chapter 7, comparing our output against a number of manually segmented ground truth masks. We show percentages of false negative and false positive errors, and discuss possible optimizations that can be applied to increase the quality. At last our conclusions are drawn and future work is presented in chapter 8.

Chapter 2

Silhouette extraction

In this chapter we look at some of the previous work done in the field of silhouette extraction. In section 2.1 we present our definition of silhouette extraction. In section 2.2, we look at different methods for background subtraction, and in section 2.3 we briefly present some other tried methods for extracting silhouettes.

2.1 Definition

Silhouette extraction is the combined process of extracting, or separating, a foreground object in video from the background, with some criteria defining what belongs to foreground and what belongs to background. This process can involve multiple steps. The result should be one or more silhouettes of people or other objects in the video that are not part of the background.

One or more silhouettes can be represented in a binary, black and white image mask, where the background and the silhouettes are black and white, respectively - see figure 2.1. This binary mask can be used for further processing by other algorithms. If there are multiple silhouettes, another reasonable representation is to give each connected silhouette a unique, positive number in the image, with black still being background.

2.2 Background subtraction

A commonly used technique for silhouette extraction is based on background subtraction [6–8]. Background subtraction is a process where the background in video is subtracted from the image, leaving us with the silhouettes of the foreground elements. A number of different methods and algorithms have been presented, as we will see below.

The basic principle of the technique is that we keep a *background model* of the video we

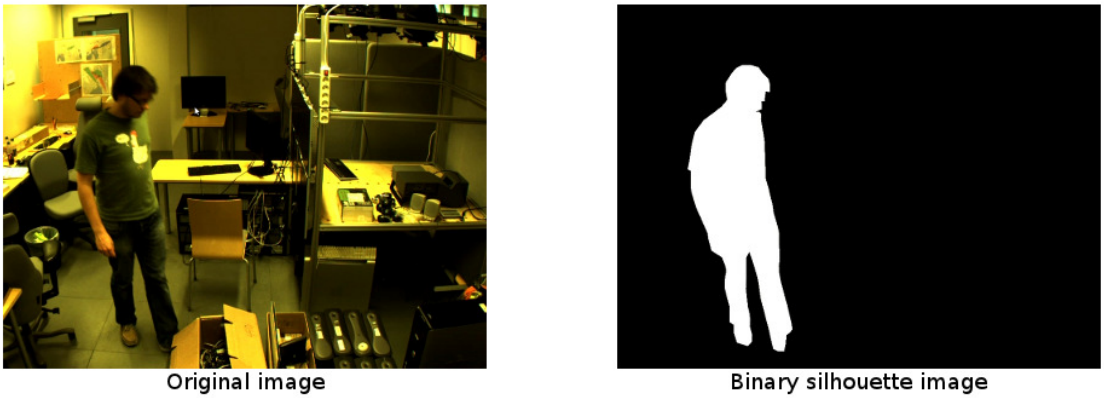


Figure 2.1: Example representation of a silhouette in a binary image

are capturing, either as a static background or as a dynamic model that is updated as the video changes. This model is then an estimate of how the background would look if no foreground objects were present [6]. We compare the current video frame against this background model, detecting foreground elements that have changed or moved in the video, or in some cases differ from a known background property, usually color.

Many different background models and alterations to this basic principle exists [7–17].

Static background

In earlier approaches to background subtraction, e.g. [18, 19], the assumption has been that the background is static.

Chroma keying, also known as green- or bluescreen is a static background subtraction technique widely used in television and movie-production. It is performed by having a solid color background (usually green or blue) in the shot, to be able to easily separate the background from the people in the foreground (see figure 2.2[1]). The background is then replaced with something else, e.g. a weather map, special effects or other videos or images. A limiting factor is that the subject in the foreground has to avoid clothes with colors that are similar to the background, as they then can be falsely segmented out as background elements.

Dynamic background

The assumption that the background is static has the effect that the algorithm won't work correctly under different changing conditions in the background video. This can for example be

[1] http://en.wikipedia.org/wiki/Chroma_key (last accessed: 15/04-2012)

Figure 2.2: Demonstration of blue screen at the Special Effects show, Museum of Science, Boston.

light conditions, e.g. light gradually changing during the day, clouds appearing over the sun, lamps being dimmed in a room etc. Moving objects in the background, like waving branches on trees, waves at the ocean, rain and snow, or bigger things like someone parking a car in the scene would also cause problems, as these elements would be recognized as foreground. This limits the usage of these kinds of algorithms to controlled environments.

Dynamic background models use different methods to keep and update data about the background over time, to be able to adjust to changing conditions in the background. The background model usually use some sort of statistics of the previous background frames. This could be e.g. the average or median of the last N frames, updating each pixel in the model on each

new frame. The update will only happen if the pixel is detected as background.

Hedayati et al. [9] and Benezeth et al. [8] reviewed some of the most common algorithms using dynamic background modeling to keep track of changes in the background.

Median based modelling is a widely used technique, having the advantages of fast computational time, and reasonably good results. *Median Filtering* extracts background depending on the median values of pixels in a buffer. Cucchiara et al. [10] showed that using median filtering allowed fast detection of moving objects. Hung et al. [20] presented an approach for reducing median computation, by using repetition checking of consecutive frames, showing good experimental results. A recursive version of Median based modelling, *Approximate median*, proposed by McFarlane and Schofield [11], tries to predict the median values to reduce computational time.

Gaussian based modelling models each pixel as one or more Gaussian distributions. It is a recursive technique, and the two most common approaches are *Gaussian Mixture Model (GMM)*, proposed by Friedman and Russell [12] and *Running Gaussian Average (RGA)* by Wren et al. in [13]. Gaussian based modelling has the advantage that it does not have to keep a large buffer of frames in memory, thus lowering the total memory requirements. Stauffer and Grimson [14] used a GMM to model each pixel, with good results in both outdoor and indoor scenarios. The algorithm takes care of slow lighting changes, flickering computer screens, waving branches on trees, etc. Shimada et al. [15] presented an approach where the number of Gaussian distributions modelling each pixel can be increased and decreased dynamically, giving reduced computational time, and higher accuracy.

Kernel Density modelling or *Kernel Density Estimation (KDE)*, presented by Elgammal et al. [16] is a non-parametric, distribution free model, estimating pixel intensity density from sample history values. Compared to a GMM-algorithm with the same amount of memory available, higher sensitivity is reported on their KDE-algorithm. Zhang et al. [17] presented a modified KDE-algorithm that also takes neighbouring pixels into consideration when estimating.

Thresholding

Thresholding is used when detecting changes in a new frame, compared to the background model. Usually, the background model is subtracted from the value of the new frame, resulting in the difference between the new frame and the background. One or more threshold values are then used to separate the background from the foreground e.g. by saying that values above a given threshold is foreground, below is background.

If thresholds are set correctly, they can improve detection accuracy, and lower computational time. A manual threshold will not adapt to changes in the environment such as change in

lighting, and therefore an automatic thresholding algorithm is the preferred choice [9]. Otsu Rosin [21] and Sezgin and Sankur [22] reviewed different methods for automatic thresholding, regarding different background models.

Data validation / refinement of foreground

Data validation or refinement-techniques take care of correcting misclassified pixels after background subtraction has been performed. With misclassified pixels we mean background pixels classified as foreground or foreground pixels classified as background. As there are usually some errors in the detection and thresholding process, this step is needed to get more accurate silhouettes.

Various techniques have been developed to take care of these errors: noise removal, blob processing, object level feedback, saliency test and optical flow test [9], to mention some. We will look closer at some refinement techniques in chapter 4.

2.3 Other ways to extract silhouettes

Even though background subtraction is a common approach for extracting silhouettes, other approaches can be used. Chen et. al. [23] used motion detection and graph cuts for accurate extraction of silhouettes, giving good experimental results in complex scenes.

Alahi et. al. [24] used a camera pair and stereo mismatch in combination with a Total Variation framework to produce a dense disparity map for extracting silhouettes. Experiments showed good results in complex scenes and with sudden changes in illumination.

2.4 Summary

In this chapter we have looked at silhouette extraction in video and presented our definition of what it is. We have looked at a common approach using background subtraction, thresholding and data validation/refinement, and finally mentioned a couple of other methods from literature. In the next chapter we look at using GPUs for parallel programming, presenting the OpenCL framework and common optimization strategies.

Chapter 3

Graphics Processing Units and OpenCL

In this chapter we look at using **Graphics Processing Units (GPU)** for parallel processing.

3.1 Introduction

In section 3.2, we look briefly at the evolvement of GPGPU. Section 3.3 introduces the OpenCL framework for development on heterogenous platforms, with our focus being on GPUs. This is followed by section 3.4, where we present some common optimization strategies used when programming on GPU.

3.2 General-purpose computing on graphics processing units

General-purpose computing on graphics processing units (GPGPU) has become increasingly popular the last few years. Frameworks like CUDA and OpenCL have made it easy for anyone with a modern graphics card to develop applications that take advantage of the great parallel processing power provided by GPUs. In the field of computer vision, this has given us the possibility to explore real-time computation of high resolution images and video, opening for lots of new applications and use-cases. In this thesis we will use the OpenCL framework for implementing our application on GPU.

3.3 OpenCL

3.3.1 Open standard for parallel programming

Khronos group, the technology consortium behind OpenCL, describe it like this:

> OpenCL™ is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL (Open Computing Language) greatly improves speed and responsiveness for a wide spectrum of applications in numerous market categories from gaming and entertainment to scientific and medical software. [5]

In other words, OpenCL can be used on multiple platforms and with various types of hardware, like CPUs and GPUs, as long as the vendor has implemented support for it. This means that we can write software that is possible to run on different hardware, as long as it has support for OpenCL.

Our focus will be on GPU implementations, even though OpenCL also can be used for different processors. As we mention in 3.3.2, different hardware may need different optimization techniques, so even though an application will run on a GPU from another vendor, it may need some vendor-specific tweaks to get good performance.

3.3.2 OpenCL-implementation and hardware used

Each vendor supporting OpenCL has its own implementation, supporting its hardware. These implementations can differ a bit, both because they are implemented for different hardware, but also because the implementations are not always identical. This means that optimizing for one vendor's hardware will not necessarily give the same results for another.

We will in this thesis use NVIDIA GPUs, and use the NVIDIA OpenCL 1.1 CUDA implementation described in the *NVIDIA OpenCL Programming Guide* version 4.1 [25]. The specification for OpenCL 1.1 is described in the *OpenCL Specification* version 1.1 [26]. The hardware used is a NVIDIA GeForce GTX 460 card with compute capability 2.1.

Most of what is described here will apply to e.g. AMD GPUs, but may also differ in certain ways regarding optimizations, implementation, etc. Covering two or more GPU vendors, and optimizing differently for both of them is beyond the scope of this thesis.

3.3.3 Kernels

In OpenCL, the functions or programs that execute on GPU (or other processors) are called *kernels*. They are written in the OpenCL C [26] language, very similar to regular C, with some

limitations, such as lack of recursion.

3.3.4 Kernel execution

NDRange

The kernels are executed in parallel over a specified number of work items, an *NDRange*, or N-dimensional range, that can be 1, 2 or 3 dimensions. As an example, if we have a kernel that is supposed to do something to every pixel in an image of size 256x256, we can launch a 2-dimensional range with width and height of the image. We then have one work item for each pixel, addressable by its x and y coordinates in the image.

Work group

The *NDRange*, described above, is divided into work groups of a size chosen by the user, or automatically by OpenCL. We could as an example choose a work group size of 16x16 for the abovementioned 256x256 NDRange, giving us 256 work groups with 256 work items in each group. Manually choosing the work group size is often advisable, in many cases even necessary, giving us more control over how the kernel is executed.

This is essential when dividing our problem, and different work group sizes can also give different performance because of memory access, as described in 3.4.2. Work items within a work group can share local memory between them, giving fast access to neighboring work items' data. They can also synchronize, meaning that each work item can do some work until a set point, then wait until every work item is finished, before continuing execution.

Work item

A work group is divided into *work items*, also called threads. Each work item executes the same kernel, but has its own unique id, both a global id, and local (work group) id, in the dimensions it is executed. This identifies the work item in the NDRange, thus giving it a way to index the data it reads and writes to.

Work item execution

The work groups are scheduled and executed in a group of 32 work items, also called a *warp*, or *wavefront* in AMD terminology. This should not be confused with a work group, where the work items can share local memory. All the work items in a *warp* start executing at the same time.

If the work items in the warp perform the same instructions, they will execute simultaneously. If conditional branches like if-statements cause work items to execute different paths of the code, each path is executed serially, making the other work items wait until they share the execution path again. For the best possible performance, branching should be avoided if possible.

3.3.5 Synchronization

As mentioned, the work items can synchronize, meaning that they at specified points in the code can wait for all the other work items to finish, before continuing execution. This can be very useful in many situations. E.g. if each work item calculate something that is going to be reused by other work items later, making sure it is written back to memory before continuing execution is essential to get correct results.

3.3.6 OpenCL memory types

There are different memory types available in OpenCL, with different sizes and properties. We will describe them below, and briefly mention possible uses where they fit.

Global memory

Global memory is located in device memory. It is accessed by memory transactions of size 32-, 64- or 128-byte. As an example, this means that if one work item reads one 4 byte integer, a 32-byte memory transaction has to be performed.

In a warp, these memory accesses are coalesced, meaning that if multiple work items in a warp are reading elements from within the same 32, 64 or 128 byte memory block, these reads are combined. This means that if the data the work items read is sequential (e.g. each of the 32 work items in the warp reads 4 bytes of sequential data), one 128 byte read can be performed. If, however, the data is very spread or the access pattern is not sequential, multiple memory accesses have to be performed. This will result in reading of lots of data that is not used, thus causing the memory performance to decrease.

Using coalesced global memory is described more in 3.4.2.

Local memory

Local memory, called *shared memory* by NVIDIA, is fast on-chip memory accessible to every work item in a work group. It is divided into memory banks, and different banks can be accessed simultaneously.

Local memory is fast for all work items in a warp, as long as *bank conflicts* — accesses to *different data elements in the same memory bank at the same time* — is avoided. Multiple work items accessing *the same elements of the same bank* have no performance decrease, as the reads are then broadcasted to the requesting work items. Writes to the same element are performed by only one of the work items - which one is undefined.

If bank conflicts between two or more work items occur, those accesses are serialized, decreasing performance.

Local memory can give a very good performance increase when used on kernels where work items are dependent on neighboring work items' data. Each work item will read in its respective data from global memory into local memory. The work group is then synchronized, before the processing starts.

Private memory

Private memory is the memory used for variables in each work item. It can not be accessed by other work items.

The private memory size and location is not defined in the OpenCL specification, so it can differ between different devices and compilers. Minimizing usage of private memory can therefore be a good precaution, but only proper profiling of the application can give an answer to the effects of this, and the maximum amount that can be used before performance decreases.

Constant memory

Constant memory is, as the name tells us, read-only memory defined on the host before execution of a kernel. It is placed in device memory, but is cached in the constant cache of a multiprocessor, giving faster memory access on a cache hit. On a cache miss, the access time is the throughput of device memory [25].

Texture (image) memory

Image2D and Image3D are special image types in OpenCL, optimized for image processing. Multiple image formats with different data types and sizes are supported, ranging from 4 to 1 channel images. We can, however, put any data we want into image objects, as long as we make sure to use the same data types (and an image format supporting that data type).

Texture memory is used for accessing Image2D and Image3D-objects in OpenCL. It is cached, meaning that on a cache miss, it has to be read from device memory, but on cache hit, it uses the texture cache, giving higher performance.

The cache is optimized for 2D spatial locality, giving best performance if work items in the same warp all read data that is located close in memory.

Other reasons to use Image2D and Image3D objects in OpenCL include:

- Addressing calculations are performed by dedicated hardware
- Image boundary checks can be ignored, returning specified data on boundary misses
- Conversion between certain image types
- Built-in optional conversion and normalization of values

Even though there are multiple advantages of using Image objects, they can only be either read *or* written within a single kernel call. This means that kernels performing changes to the data during the execution will have to store these changes in other types of memory during execution, and write them back to another write-only Image object in the end. In these cases, using global memory with proper coalesced access may be a better choice.

3.4 Optimization strategies on GPU

There are multiple well-known strategies for optimizing performance when programming on GPUs. Limits in memory sizes, different types of memory and knowing the way the work items on GPU accesses memory are important aspects.

Another important aspect is to think about what kind of optimizations to focus on in the different applications. Focusing on memory access in an application that mainly has problems with branching, is for instance, not the best approach. Of course, some applications might need more than one optimization strategy, but the main point is to focus on the major issues first.

In this section, we present some optimization-strategies often used when implementing on the GPU.

3.4.1 Keeping and reusing data in device memory

Memory transfers between the host (CPU) and the GPU are very expensive, and can in many cases take more time than the actual processing done in the kernel on the GPU.

As we usually call multiple kernels that perform different operations to the same data, transferring the data between host and device memory between every kernel call will cause a fatal blow to the performance. Keeping the data in device memory after it has been transferred the first time, and minimizing transfers back to the host unless it is absolutely necessary is of major

importance. This way one kernel performs a task, and the next one will reuse its memory as input — without any transfers back to the host. The last kernel in the pipeline will finally transfer back the needed results to the host.

Reorganizing our algorithms and kernels to reuse memory as efficiently as possible can therefore be a major optimization.

3.4.2 Coalesced memory access

Accessing global memory is one of the biggest bottlenecks in GPU programming. When reading and writing to global memory inefficiently, we can get major performance drawbacks compared to the potential throughput.

As we mentioned in 3.3.6, each warp coalesces memory accesses, meaning that if each work item in a warp read successive memory elements, they will be combined, or *coalesced*, into one or more reads. If they read completely different memory locations, we will in the worst case get 32 accesses of 32 Byte (when each work item reads 4 bytes).

Knowing that one 128 Byte memory access would be sufficient if all the elements read were coalesced (in the best case), it is clear that planning the layout of memory and work groups is very important.

3.4.3 Local memory

Accessing neighboring elements

When work items are dependent on accessing neighboring work items' values, using local memory is essential. Each work item in a work group will be responsible for reading its own value from global into local memory. The work group is then synchronized, making sure all the values are available before execution.

After this is done, the work items access the values from local memory instead of global, allowing faster access, unless bank conflicts occur, as described in 3.3.6.

3.4.4 Read-only data in constant memory

Some applications use data that is read many times, but never changes. In some of these cases, using constant memory can give us a good performance boost. In some cases, however, the data is too big to fit into constant memory, therefore it has limited uses.

One example where it would be smart to use constant memory, is if we are performing some filtering to an image, using a filter-kernel. The filter-kernel is usually a block of values, maybe

8x8 or 16x16 pixels in size. The block is read for every work item in the whole image, and does not change during execution. Mapping this region into constant memory could be very profitable, as the memory is cached in the constant cache, thus giving better performance.

3.4.5 Using OpenCL Images

As mentioned in 3.3.6, using the OpenCL Image types can be profitable in many cases. In kernels that need a lot of boundary checking, using images can be a good choice, as they can be read out of bounds without crashing our program, but rather returning a border value, usually 0.

This way, branching caused by boundary checking can be avoided, increasing the execution efficiency of the warps.

3.5 Summary

In this chapter we have presented the OpenCL framework, and how it can be used for GPU programming. We have looked at the different memory types, execution model, and presented some common optimization strategies to keep in mind when implementing on GPU.

In the next chapter we present a silhouette extraction algorithm, explaining the different steps and talking a bit about how to implement it on GPU.

Chapter 4

Silhouette extraction algorithm

In this chapter we look at an algorithm for silhouette extraction proposed by Kim et. al. in [27]. This algorithm is the basis for our implementation-work in this thesis, implementing parts of it on GPU for the best possible execution time.

We present the algorithm as it is in [27], looking into the details of the different parts. We also discuss briefly some possible optimization techniques for implementing the different parts of it on GPU. Our actual GPU implementation is presented in chapter 5.

4.1 Introduction

In the paper "Robust Foreground Extraction Technique Using Gaussian Family Model and Multiple Thresholds" [27], Kim et. al. describe a full pipeline for extracting silhouettes from video. The algorithm is adaptive to changes in light and changes in geometry over time.

The steps covered in the paper include choosing a statistical model based on training data (the first N frames in the video), creating and maintaining a background model based on this data, and extracting the foreground using multiple thresholds.

It also includes shadow removal and various morphological processes for improving and refining the extracted foreground mask, giving a better result in total.

In the following sections we take a closer look at the different parts of this algorithm, and how they are connected. We will look at the following:

- Section 4.2: Training step
- Section 4.3: Background modeling
- Section 4.4: Background subtraction, thresholding and foreground extraction
- Section 4.5: Foreground refinement / morphological processes

- Section 4.6: Profile extraction
- Section 4.7: Background update
- Section 4.8: Output format

4.2 Training step

The algorithm has a training step initially, where the N first frames of the video are used to get data about the background in the video, and choose a background model based on these data. The number of training frames, N is not defined, but chosen by the user.

4.2.1 Selection of distribution form

Different probability distributions have different properties, and are often used and combined in certain ways when creating background models.

This algorithm uses the **Generalized Gaussian Family (GGF)** distributions for modeling the background. The GGF-model is parametric, meaning we can choose which form of distribution to use for each pixel. This gives us the ability to choose the model that is most fit for our data. In this specific algorithm, Gaussian and Laplace distributions are used, but this could be expanded easily by changing the shape parameter.

Excess kurtosis

To be able to decide which distribution to use for each pixel of the image, we have to calculate the excess kurtosis of each pixel of the training data. The excess kurtosis tells us if the data is flat or peaked relative to a normal distribution, giving us information about the most suitable model for each pixel. For calculating the excess kurtosis we first need to calculate the mean and variance of the traning data.

The equation for calculating excess kurtosis, $g2$, is shown in eq. 4.1. Here, n is the number of samples/training images. x is the pixel value and μ is the mean of the samples for one pixel.

$$g2 = \frac{m4}{\sigma 4} - 3 = \frac{n\sum_{i=1}^{n}(x_i - \mu)^4}{(\sum_{i=1}^{n}(x_i - \mu)^2)^2} \tag{4.1}$$

For each pixel, we select the distribution form most fit for its excess kurtosis value, and use that distribution when creating the background model. The excess kurtosis for Gaussian is 0 and for Laplace it is 3. In the GGF-model, the shape parameter β selects the distribution form.

$\beta = 2$ means a Gaussian distribution, and $\beta = 1$ means a Laplace one. These values will be used when creating and updating the background model, described in section 4.3.

4.2.2 Possible optimizations

Memory usage when calculating statistics

For calculating the excess kurtosis, we need the mean and variance of the data. As we see in eq. 4.1, we also have to sum N training frames per pixel, meaning we have to keep all the N training frames in memory to perform the needed calculations. This can consume a lot of memory, depending on the size of the video, and the number of training frames.

As memory is more limited on GPU than on the host, and memory transfer times gives us longer total execution time, we propose to use *running*, or *online* statistics, as presented in [28] and [29]. Instead of keeping all frames, we update the mean and variance when each new frame is processed, keeping only the result.

We will investigate this futher in chapter 5.

4.3 Background modeling

A background model is, as described in chapter 2, used for keeping statistics of the background over time. This way we can compare new frames to the background model and use this information for segmenting out the foreground elements.

4.3.1 Two background models

The algorithm uses two distinct background models: one based on the luminance of the image and one based on the color.

As normal RGB components are sensitive to changes in lighting and noise in the video, the luminance model is used for the initial detection and segmentation of the foreground objects.

The color component on the other hand, is used for removing shadow regions incorrectly detected as foreground in the previous step. More specifically, the H component of the *HSI* color space is used. The color component is used for shadow-removal because even though the brightness change in shadow-regions, the color remains rougly the same.

4.3.2 Color conversion

The assumed input to the algorithm is RGB image data. As we need one luminance component and one color component for our background models, the RGB-data is converted to these two components, as described below.

Luminance component

The equation for extracting the luminance component *Y* is not mentioned by Kim et.al., so we extract it using eq. 4.2, assuming a regular conversion from RGB to YUV [30].

$$Y = R \times 0.299 + G \times 0.587 + B \times 0.114 \tag{4.2}$$

Color component

The color component *H* from the HSI-model is used for the color model. This is extracted as shown in 4.1 [27].

$$\begin{aligned} I &= \max(R, G, B) \\ S &= \begin{cases} (I - \min(R, G, B)/I) & if\ \ I \neq 0 \\ 0 & otherwise \end{cases} \\ H &= \begin{cases} (G - B) \times 60/S & if\ \ I = R \\ 180 + (B - R) \times 60/S & if\ \ I = G \\ 240 + (R - G) \times 60/S & if\ \ I = B \end{cases} \\ & if\ \ H < 0\ \text{then}\ \ H = H + 360 \end{aligned}$$

Figure 4.1: Extraction of the color component H from RGB

4.3.3 Creating background models

When we have performed the training step and chosen distribution types, we can create the actual background models, starting with a new video frame. The equation for GGF-distributions is shown in eq. 4.3 where Γ is a gamma function, and $\gamma = \frac{1}{\sigma}(\frac{\Gamma(3/\rho)}{\Gamma(1/\rho)})$. x is the pixel value, σ is the standard deviation of the pixel value and μ is the mean. The mean and standard deviation is first calculated from the training frames, and later updated as shown in section 4.7.

The value of ρ choose the distribution type, so $\rho = 2$ represents a Gaussian distribution and $\rho = 1$ a Laplace distribution.

$$p(x : \rho) = \frac{\rho\gamma}{2\Gamma(1/\rho)} exp(-\gamma^{\rho}|x - \mu|^{\rho}) \tag{4.3}$$

This is done for both color data and luminance data, creating two distinct models. The same equation is then used for each new frame.

4.3.4 Possible optimizations

Color-conversion

The algorithm assumes RGB image data, but converts it to Y (luminance) and H (color). Getting YUV-data as input instead of RGB could save at least one conversion, getting the Y-component directly. This might save some computation time, but this also depends on the possibility of conversion from YUV to HSV.

4.4 Background subtraction, thresholding and foreground extraction

4.4.1 Background subtraction and thresholding

In the thresholding step, three different threshold values, $K1$, $K2$ and $K3$ are used to define which group each pixel of the subtracted image belong to. The pixels are divided into four groups:

- Reliable background
- Suspicious background
- Suspicious foreground
- Reliable foreground

The threshold values *K1, K2, K3* are predefined, and needs to be tuned to perform well on different video data. This is a limitation of this algorithm, making it a bit harder to adapt to new locations and environments.

The background is subtracted and pixels are thresholded as shown in 4.2 [27]. L_I is the luminance component of the current frame and L_B is the luminance component of the background

model. σ is the standard deviation of the background model and p means the current pixel. The luminance background-model is subtracted from the luminance data of the new frame, and the absolute value of this subtraction is used as input to the thresholding.

$$BD(p) = |L_I(p) - L_B(p)|$$

$$\begin{cases} BD(p) < K_1\sigma(p) & \Rightarrow \text{(a) Reliable Background} \\ K_1\sigma(p) \leq BD(p) \leq K_2\sigma(p) & \Rightarrow \text{(b) Suspicious Background} \\ K_2\sigma(p) \leq BD(p) \leq K_3\sigma(p) & \Rightarrow \text{(c) Suspicious Foreground} \\ K_3\sigma(p) \leq BD(p) & \Rightarrow \text{(d) Reliable Foreground} \end{cases}$$

Figure 4.2: Subtracting background and thresholding image

Pixels that belong to the *Suspicious foreground* region will sometimes be shadows and reflections incorrectly detected as foreground. Therefore, a shadow removal is performed on these pixels, using the equation in 4.3 [27]. Here, H_I is the color component of the image, H_B is the color background model and σH is the standard deviation of the color background model.

The pixels that are marked as being shadows are relabeled as *Suspicious background*.

$$\begin{aligned} if \ \ & \big(p \in region(c) \& |H_I(p) - H_B(p)| < K_1\sigma_H(p)\big) \\ & then \ p \Rightarrow region(b) \end{aligned}$$

Figure 4.3: Shadow removal in suspicious foreground

4.4.2 Foreground extraction

After the thresholding has been performed, the pixels identified as *Foreground* and the remaining pixels in *Suspicious foreground* are put into a one channel image as foreground pixels *(1)*. The rest are all marked as background *(0)*. In addition to this, we keep the pixel classifications, as they are to be used in 4.6.2.

4.4.3 Possible improvements

To get more control over the shadow removal, introducing another threshold value, e.g. KBG instead of using $K1$ in this step would allow for more fine-tuning of the result.

4.5 Foreground refinement / morphological processes

4.5.1 Motivation

After foreground extraction is performed, the foreground may contain gaps and holes wrongly detected as background, or noise wrongly detected as foreground. As this is not necessarily good enough for many use cases, improvements of these regions are needed. The resulting one channel image from the foreground extraction step in section 4.4 is further processed using various morphological processes.

4.5.2 Closing and opening

Closing and opening operations are performed to remove small noise regions, and growing together regions that have small gaps between them. Closing and opening are morphological operations consisting of in total four operations, as shown in Figure 4.4. *Closing* is performed by doing one dilate operation, followed by one erode. *Opening* is the opposite of this, one erode followed by one dilate.

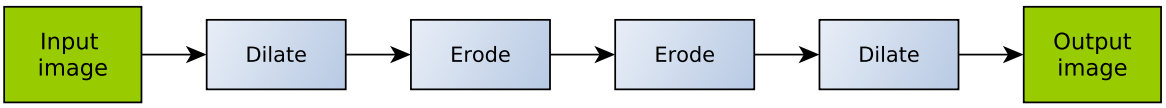


Figure 4.4: Steps in a closing and opening

Dilation

Dilation is the process of growing foreground regions. Each pixel looks at its neighbouring pixels, defined by a structuring element (most often a 3x3 block of pixels, where the given pixel we look at is the one in the middle). If one of the neighboring pixels are marked as foreground, that pixel will also be marked as foreground. This is a way to remove small holes in foreground regions, making the detected foreground more robust.

Erosion

Erosion is the opposite of dilation. We look at the neighbouring pixels, and if one of them are marked as background, this pixel is also marked as background.

Effect

The effect of a closing followed by an opening is that pixels that are close to other pixels will be merged together. Pixels that are spread around, however, such as noise in the image that is small enough, are removed. There is most often still some noise left after this is done, if there are larger noise regions in the image.

Figure 4.5 shows the effect of a closing and opening of a segmented video frame. As you can see in the first dilate step, many pixels of the segmented person in the foreground are tied together. In the two following erode steps, some of the smaller noise regions are removed. A negative side effect of this is that some of the unconnected regions inside the person are also removed.

The last dilate step expand the remaining foreground regions, also including the remaining noise regions.

4.5.3 Connected Components Labeling

Some regions are wrongly detected as foreground, even after the closing and opening step explained above. To filter out these regions, we use a threshold TH_{RG}, set to 0.1% of the image size. We remove the regions that have fewer connected pixels than TH_{RG}. To be able to do this, we have to find the connected pixels, or *components* of the image, using a **Connected Components Labeling (CCL)** [31] algorithm.

Input and output

As input, we use the output of the closing and opening step, where background pixels have the value *0* and the foreground pixels have values *1*. As output, a one channel image where each region of connected components have a unique positive integer is preferred. We also need to store the size of each region.

Choice of algorithm

There exists multiple algorithms for CCL, both on CPU [32–34] and GPU [35–37]. Kim et al. [27] use one of the algorithms from [38], but do not mention which one. Since we are focusing on a GPU-implementation, we have to find one that fits our goal of high performance.

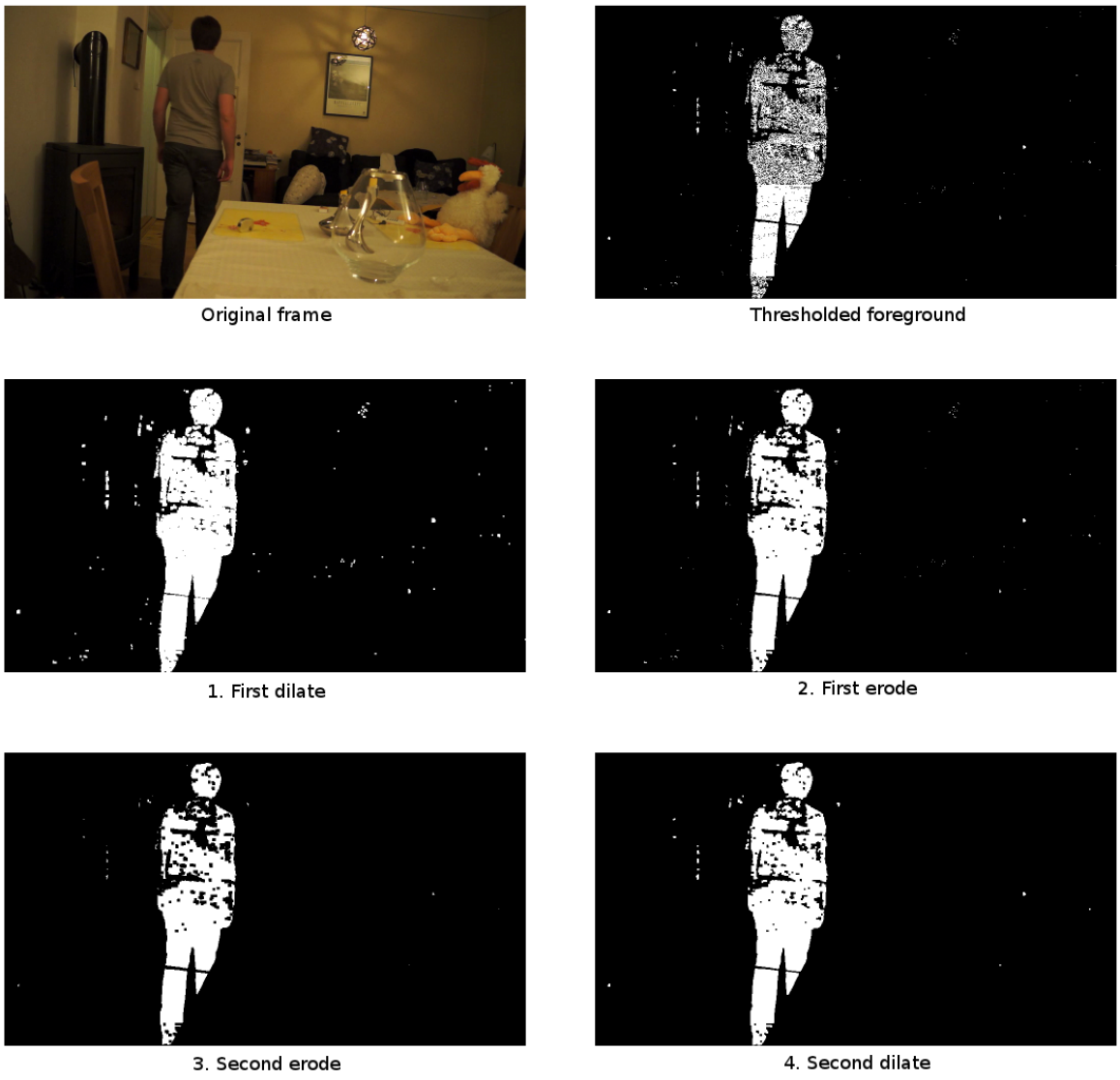


Figure 4.5: Effects of a closing and opening

4.5.4 Optimization strategies

When implementing these morphological operations on GPU, we have to think about how the memory is accessed. In the previously mentioned steps, each pixel only depended on itself. Here we have to look at neighbouring pixels, and as we saw in chapter 3, using local memory could be one way to optimize this. As we also have to check image boundaries when accessing neighbouring pixels, using the OpenCL Image types might prove helpful. OpenCL Images have automatic boundary checking, meaning we can avoid branching that would normally be caused by checking image boundaries.

The Connected Components Labeling is even more demanding, potentially accessing pixels spanning over large areas of the image. OpenCL Images, local memory and merging the data between the work groups are all possible strategies to look into, depending on the algorithm.

4.6 Profile extraction

The last steps consists of smoothing the resulting foreground regions, using a profile extraction technique that covers small concavities and holes inside and around the objects.

4.6.1 Covering the object

It is performed by moving a one pixel thick drape from one side to the other of each object. The pixels adjacent to the drape are connected with something described as an "elastic spring", covering the object, but without infiltrating gaps smaller than a set threshold M. This is performed from all edges of the object, covering the object as shown in Figure 4.6. The foreground is shown as black in the figure, for better readability.

After this is done, everything inside the "elastic spring" is marked as foreground, refining the silhouette.

4.6.2 Fixing covered holes in the object

This profile extraction step covers actual holes inside the objects, so as a final step, holes in the objects that are bigger than a set threshold and that was masked as *reliable background* in the thresholding step are opened again.

4.6.3 Optimization strategies

To limit our scope a bit, we will not implement this specific part on GPU. We will however discuss a bit what to think about if doing so.

4.7 Background update

The algorithm supports two different kinds of background changes:

- Gradual changes in lighting
- Changes in background geometry

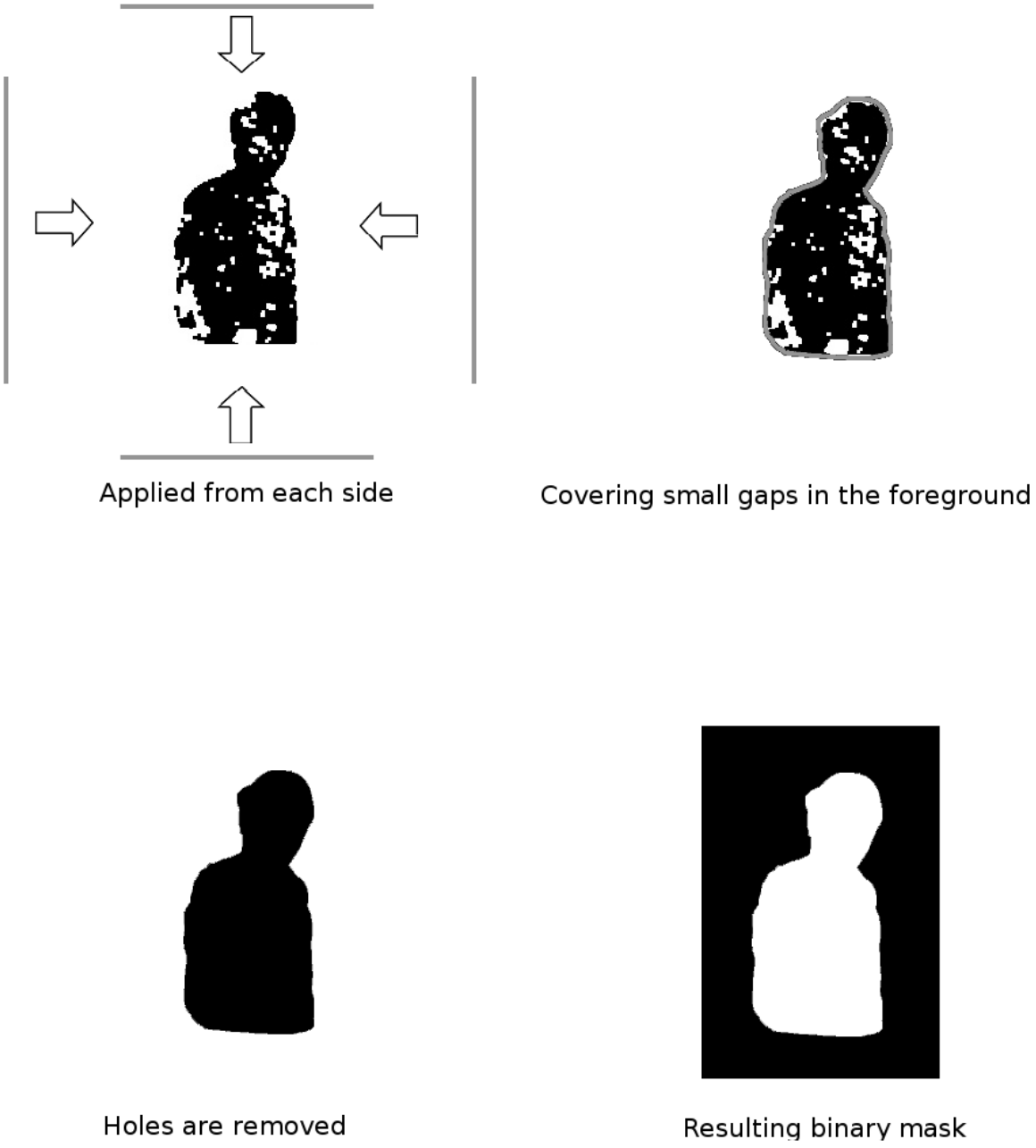


Figure 4.6: Profile extraction applied to foreground element

4.7.1 Gradual changes in lighting

To deal with the gradual changes in the background caused by changes in the lighting, the mean and variance used in the background model is updated with a running average for each pixel, as

shown in eq. 4.4.

$$\mu_t + 1 = \alpha x_t + (1 - \alpha)\mu_t$$
$$\sigma^2_{t+1} = \alpha(x_t - \mu_t)^2 + (1 - \alpha)\sigma^2_t \tag{4.4}$$

Here, x_t means the current pixel value. $\alpha = 0.5$ if x_t is marked as background in the previous steps. If x_t is foreground, $\alpha = 0$. This means that the background will only be updated for the pixels marked as background in the preivous steps. The foreground pixels will not alter the background model.

4.7.2 Changes in background geometry

If an object in the video is moved to a new location, and stays there for a long time, or permanently, both the new and the old location will be detected as foreground. Since it will not change over time, it will be marked as foreground permanently.

This problem is handled by having a counter on foreground pixels. If they have been foreground for TH_{Bg} frames, the background models are replaced by new ones, thus setting the foreground pixel as background again. The threshold TH_{Bg} is chosen manually, depending on the wanted behaviour.

The replacement is only done if the pixels are not connected to a foreground region where some of the pixels change over time. E.g. if a person is standing in the same spot, but moving his arms, it would be wrong to set the rest of the body as background, just because only the arms are moving. Therefore, if any pixels in the same connected region are changed, the counter for all the pixels in the region are set to zero again.

Memory requirements

To be able to replace background models at any given time, we need to keep image data for a certain number of frames at all times, so we can recalculate the mean and variance of each pixel when needed. As mentioned in section 4.2, we could use running statistics for keeping the mean and variance, saving a lot of memory.

This means we will keep another set of mean and variance, separate from the background model, in case we need to replace the background model for some of the pixels.

4.8 Output format

The output of a silhouette extraction algorithm should obviously be silhouettes in some form. As the term silhouettes can be a bit abstract, we want to clarify in what form we will output the silhouettes.

4.8.1 One number for each silhouette

Different applications will probably benefit from different kinds of output, but with this approach, the output silhouettes are represented as positive numbers in a one channel image. Each unique number represents one silhouette of connected pixels, and the number *0* represents background pixels.

4.8.2 Limitations

The silhouettes can be sorted and relabeled so that e.g. the biggest silhouettes gets the lowest number. This does however not guarantee that a silhouette will have the same identification number over multiple frames, as the size will change from one frame to the other. Therefore, we would need other measures for identifying and following objects. This will not be explored further in this thesis.

4.9 Summary

In this chapter we have explored an algorithm for extraction of silhouettes from video. We have looked at the different steps, and briefly discussed strategies for implementing it on GPU.

In chapter 5, we look at the implementation of some of the specific parts of this algorithm on GPU, using OpenCL. We try out different optimizations and measure the effects they have on the execution time.

Chapter 5

GPU Implementation

In this chapter we present a GPU implementation of most of the steps in the algorithm we presented in chapter 4. We try out various optimization strategies and evaluate the effects.

5.1 Introduction

As we saw in chapter 3, there are several ways to optimize the performance when using GPU implementations in OpenCL. In the following sections, we look at the steps of the algorithm, as presented in the previous chapter, and try different optimization approaches to improve the execution time.

We describe our implementation and try out different solutions and alterations.

5.1.1 Steps not implemented

To limit the scope of our work a bit, the profile extraction and some of the filtering steps are not implemented, but we present some aspects to think about when solving these steps on GPU.

5.1.2 Method for testing and evaluation

As optimizations are explored, we measure the execution time of each step by taking the current time before and after the execution and calculating the elapsed time.

The results shown is to be read as the runtime of one video frame, showing the *minimum*, *maximum* and *mean* runtimes, as this can differ between frames. In the case of connected component labeling, the differences between frames can be pretty big, as they depend on the detected foreground in the image, and size of the connected components.

Running of tests

Tests are run and results calculated as follows:

- All the tests, on two different videos, resolutions and variations of the algorithm, are run 5 times each.
- For the mean-result, the average of the 5 runs is presented.
- For the min-result, the min of all 5 runs is presented.
- For the max-result, the max of all 5 runs is presented.

Total runtime measurement

After finding the implementations that give the best runtime of each step in the algorithm, we finally measure the total execution time of all the implemented steps. The maximum execution time has the highest weight of the measurements here, telling us what we can expect from the algorithm in the worst case (with our video data).

Experimental data used for testing

The video we have used is captured with a *Sony NEX-5N* digital camera. The original captured data was 1920x1080 pixels, but we have resized it to two sizes: 1024x576 and 512x288, reencoded to x264, 25 FPS. All test video was encoded in the same way - no noise reduction was applied.

We use two videos for our experiments. What will be referred to as “Video 1” is captured in a living room at night. The light sources in the room are all artificial (lamps), creating different shadows as a subject walks around in the room. There are also pictures on the walls with glass frames, causing reflections.

“Video 2“ is captured in larger room with chairs and desks, some glass walls and polished floors, possibly causing reflections. A person walks around in the scene, sometimes being partly covered by the furniture in the room.

Both videos start without a subject in the foreground, as described in 5.1.3. The first frame of the videos are shown in fig. 5.1 and fig. 5.2.

Figure 5.1: Video 1 - First frame

Figure 5.2: Video 2 - First frame

5.1.3 Assumptions and notes

Video dimensions

We assume that the video width and height is dividable by 16 and 32 as this is needed by our chosen work group sizes. A solution to this limiation could be to pad the input video to the nearest width and height, and not include these padded regions on output, but we have not implemented this.

Empty background initially

Another assumption about the video is that it starts with a few seconds of just background, without foreground elements present. This is needed by the training step of the background subtraction algorithm we use, to build an initial background model.

Different video, different results

In some of the steps in the program the content of the video will have an impact on performance. E.g. a video with multiple, large foreground objects, will give a longer runtime than a video with few, small foreground objects. This is especially the case in the CCL part of the algorithm, where the number of iterations is dependent on the size of the connected component.

Even though our test video try to cover both large and small objects in the foreground to give a realistic runtime measurement, the results will not automatically apply to any video.

5.1.4 Choice of work group sizes

We have chosen to try two different work group sizes: 16x16 and 32x32 work items in each. We do this to see if it affects the runtime, and in that case, how much. In addition to this, in the kernels that do not require fixed work group sizes (where each work item can work independently), we also let the OpenCL subsystem choose its own work group sizes (by not specifying it explicitly). This is done to see if it results in better or worse runtimes than our chosen work groups.

5.1.5 Source code

The source code for our implementation can be downloaded from http://hjem.ifi.uio.no/kristhk/master-thesis/. The code is released under the *GNU Lesser General Public License Version 2.1 (LGPLv2.1)*.

5.2 Training step

5.2.1 Parallelization

The training step, where we calculate the mean and variance of a given number of frames, is per-pixel parallel. This means that each work item can be responsible for its own pixel without needing to access any of the other work items' data. This makes the implementation of the traning step very easy, not having to think about memory accesses between neighbouring threads.

The training step also includes a kernel that performs conversion of RGB data to luminance Y and color component H. The measurements performed are of both these kernels in total.

We run the kernels as a 2D-grid with the width and height of the input image. The work group sizes are as described in 5.1.4.

5.2.2 Running statistics

As mentioned in chapter 4, we use running statistics for calculating mean, variance and excess kurtosis. Instead of grabbing all training frames, transferring them to the GPU-memory, and then executing the training-kernel, we execute the kernel once for each training frame, copying only that frame.

The amount of data transfer is the same in total, as we execute the kernel for every new training frame. We can, however choose how many training frames we want to use, without the amount of memory available on the GPU being a limiting factor.

Another problem with copying and processing all the frames in one step is that we might have to drop new frames that were supposed to be used in the next steps, because of the training step taking too much time. By processing one frame at a time, we can keep our goal of real-time performance, without dropping any frames.

5.2.3 Optimizing memory access

Since each thread only needs data for its own pixel, we will not have to use local memory shared between threads. We do, however, have to read and keep some data in global memory between each iteration of the training step:

- Mean
- Second, third and fourth moment about the mean
- Variance

- Image data
- Storage/map for which distribution to use

Bundling data

This data has to be stored for both the color and the luminance, creating a need for lots of data buffers to be passed to the kernel, and lots of reads and writes to global memory. To minimize this we choose to bundle / pack multiple values into one buffer, reducing the number of reads/writes we have to do in each kernel.

We keep image data in an *int2*, distribution map in an *int2*, mean and variance in a *float4*, and each of the moments about the mean in *float2*s. This way, each thread can read and write more than one value from global memory at a time.

We test our implementation with and without this data bundling, to see if we gain anything.

5.2.4 Experimental results

As we can see from the results in table 5.1, the runtime differences are quite small, actually showing a slightly worse min and mean runtime on the 512x288 videos that have bundled data. The max runtime is though around 0.22 ms lower on all workgroup sizes.

On the 1024x576 videos, we see a slightly more positive difference all over, giving around 0.15 ms lower min and mean, and on average 0.28 ms lower max runtime.

Even though the differences seem very small, we do gain a little bit on high resolution videos. The *16x16* work group size is giving the best overall performance, showing little difference in the 512x288 videos, but slightly lower (around 0.10 ms) min and mean runtimes on the 1024x576 ones, compared to the other work group sizes.

The runtime does not differ mentionable between the two videos. This was expected, since the kernels in the training step are all per-pixel parallel and the data does not change the behaviour of the program.

5.3 Background modeling

Creating and updating the background model for every new frame is a relatively simple procedure, since this is also a per-pixel parallelization. The data needed for reading and writing is bundled in the same way as mentioned above.

The kernel is very simple, just performing the needed calculations and writing back to memory. It does perform a lot of mathematical operations that could slow down the execution, but

Without bundling of data												
WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
Auto	0.38	1.01	0.42	1.69	2.45	1.78	0.38	1.04	0.41	1.68	2.57	1.76
16x16	0.39	0.99	0.42	1.59	2.44	1.68	0.39	0.99	0.41	1.58	2.44	1.65
32x32	0.38	1.02	0.42	1.68	2.52	1.78	0.38	1.05	0.41	1.69	2.81	1.76
With bundling of data												
Auto	0.44	0.79	0.47	1.55	2.27	1.63	0.44	0.76	0.46	1.55	2.23	1.60
16x16	0.44	0.77	0.46	1.45	2.22	1.53	0.44	0.77	0.46	1.44	2.19	1.50
32x32	0.46	0.80	0.48	1.54	2.30	1.63	0.46	0.79	0.48	1.54	2.30	1.60

Table 5.1: Training step: Runtime in milliseconds

no branching is needed, making this a straight forward kernel to implement. The code for the kernel using bundled memory is shown in listing 5.1. Accessing multiple elements in bundled data is done by accessing varname.s0, varname.s1 etc.

```
__kernel void update_bg_model(global int2* Data, global float2* lc_Model,
      global float4* M_V, global int2* DistMap) {
  /* id for this pixel */
  int id = get_global_id(1) * get_global_size(0) + get_global_id(0);

  /* M_V contains: luma-mean, color-mean,
     luma-variance, color-variance, in that order */
  float4 MV = M_V[id];
  float2 model = lc_Model[id];
  int2 dist_map = DistMap[id];
  int2 lc_data = Data[id];

  /* Standard deviation is sqrt of variance */
  float lStdDev = sqrt(MV.s2);
  float cStdDev = sqrt(MV.s3);
  int lDist = dist_map.s0;
  int cDist = dist_map.s1;

  /* Calculate luma model */
  float Yl = (1/lStdDev) * (pow((tgamma((float)3/lDist)) /
    (tgamma((float)1/lDist)),0.5));

  model.s0 = ((lDist * Yl) / (2*tgamma((float)1/lDist))) *
    exp(pow(-Yl,lDist) * pow((fabs((float)lc_data.s0 - MV.s0)),lDist));

  /* Calculate color model */
  float Yc = (1/cStdDev) * (pow((tgamma((float)3/cDist)) /
    (tgamma((float)1/cDist)),0.5));

  model.s1 = ((cDist * Yc) / (2*tgamma((float)1/cDist))) *
```

```
31      exp(pow(-Yl,cDist) * pow((fabs((float)lc_data.s1 - MV.s1)),cDist));

33    /* Write out model to global mem */
      lc_Model[id] = model;
35  }
```

Listing 5.1: OpenCL kernel for creating/updating background model

5.3.1 Experimental results

As we see in table 5.2, the runtime differences of bundling and not bundling the data in this kernel are very small, with only the maximum time showing a slightly, non significantly better runtime when bundling. This kernel needs fewer buffers than the mean, variance and kurtosis kernel in the training step - so a possible explanation is that unless a large amount of buffers are used, the effect of bundling the data is minimal.

Without bundling of data												
WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
Auto	0.62	0.75	0.62	2.55	2.68	2.56	0.62	0.76	0.62	2.56	2.69	2.56
16x16	0.60	0.74	0.60	2.36	2.48	2.36	0.60	0.75	0.60	2.36	2.49	2.37
32x32	0.66	0.80	0.67	2.61	2.73	2.62	0.66	0.83	0.67	2.61	2.75	2.62
With bundling of data												
Auto	0.62	0.71	0.63	2.57	2.71	2.67	0.62	0.69	0.63	2.58	2.66	2.59
16x16	0.60	0.67	0.61	2.35	2.44	2.36	0.60	0.70	0.61	2.36	2.43	2.36
32x32	0.67	0.76	0.67	2.58	2.66	2.59	0.67	0.76	0.67	2.59	2.68	2.60

Table 5.2: Background modeling: Runtime in milliseconds

5.4 Background subtraction, thresholding and foreground extraction

5.4.1 Subtraction and thresholding

The background model created in the step above is subtracted from the image data, to get the difference. Afterwards, our predefined thresholds are used to separate each pixel into the foreground and background groups explained in chapter 4. This is a pretty straight forward kernel, but it includes some if-statements, creating branching.

As we also use manually set threshold values *K1,K2,K3* we try to define them as constants instead, to see if we gain anything. We also introduce a new threshold value, *KBG*, to be able to fine tune the shadow removal independent on the value of *K1*.

The code for this kernel is shown in listing 5.2.

```
__kernel void subtract_and_threshold(global int2* Data, global float2* lcModel,
global float4* M_V, __write_only image2d_t output_img, global int* region_values,
                                    /*- If input of next kernel is glbl_mem:
                                    global uchar* output_buffer -*/) {
  int width = get_global_size(0);
  int x = get_global_id(0);
  int y = get_global_id(1);
  int id = (y*width) + x;

  /* Get data */
  int2 lc_data = Data[id];
  float2 model = lcModel[id];
  float4 MV = M_V[id];

  float lData = (float) lc_data.s0;
  float cData = (float) lc_data.s1;
  float lModel = model.s0;
  float cModel = model.s1;

  /* Subtract luma data for current frame from luma model */
  float BD = fabs(lData-lModel);

  /* Initialize testresult to 0 */
  uint outval = 0;
  int region_val = 0;
  float lStdDev = sqrt(MV.s2);
  float cStdDev = sqrt(MV.s3);

  if(BD < (K1*lStdDev)) {  // Background
    outval = 0;
    region_val = 0; }
  if((K1*lStdDev) <= BD && BD <= (K2*lStdDev)) {  // Suspicious background
    outval = 0;
    region_val = 1; }
  if((K2*lStdDev) <= BD && BD <= (K3*lStdDev)) {  // Suspicious foreground
    outval = 1;
    region_val = 2;
    //    Eliminate shadows from suspicious foreground with help from color-data:
    if(fabs(cData - cModel) < (KBG*cStdDev)) {
      outval = 0;
      region_val = 1; } }
  if((K3*lStdDev) <= BD) {  // Foreground
    outval = 1;
    region_val = 3;  }

  /* Write out which group the different pixels
     were thresholded to */
  region_values[id] = region_val;
```

```
49    /*- Depending on the next kernels input, either: -*/
      /* Write output to global memory */
51    output_buffer[id] = outval;
      /*- or -*/
53    /* Write output to Image2D */
      uint4 output_mask;
55    output_mask.s0 = outval;
      write_imageui(output_img,(int2)(x,y),output_mask);
57  }
```

Listing 5.2: OpenCL kernel for background subtraction and thresholding

5.4.2 Output data

The output from this kernel is a one channel image where the foreground pixels have the value 1 and the background pixels have the value 0. It is stored in either an OpenCL 2D-image, or a uint8_t global memory buffer, depending on the kernels used in the closing and opening step, as they expect either an image or a buffer. When writing to 2D-image, we use the *CL_R* format, meaning that we only use one channel in the image for storing data.

Since the threholding divide each pixel into one of four groups, we also store this as values 1-4 in a global memory buffer.

5.4.3 Experimental results

As we can see in table 5.3, using constant memory for the threshold values give little or no difference. The max runtimes are sometimes higher when using constants, but in some cases the min and thus mean runtimes are actually lower.

A possible explanation to this could be that on the first frame, a cache miss occurs, giving slower memory access, but on the rest of the video, the lookups will give cache hits, giving higher performance. We have, however not tested if this is actually the case. Since this kernel does not use these constants many times every execution, the benefits are probably too small to pinpoint exactly, and effects of using constants would probably be more visible in a kernel reading the same values many times on every execution.

5.5 Morphological processes

The morphological processes are the more advanced parts of this algorithm when it comes to implementation on the GPU. Since these algorithms depend on data in their pixel neighbourhood and the rest of the image, both efficient memory accesses and boundary checking are

Without thresholds as constants												
WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
Auto	0.09	0.15	0.09	0.49	0.69	0.53	0.09	0.15	0.09	0.49	0.58	0.51
16x16	0.08	0.17	0.08	0.27	0.38	0.27	0.08	0.17	0.08	0.37	0.47	0.39
32x32	0.11	0.18	0.11	0.38	0.55	0.40	0.11	0.21	0.11	0.38	0.50	0.40
With thresholds as constants												
Auto	0.08	0.20	0.09	0.38	0.54	0.39	0.08	0.20	0.09	0.38	0.54	0.39
16x16	0.08	0.19	0.08	0.28	0.43	0.28	0.08	0.19	0.08	0.28	0.45	0.28
32x32	0.11	0.23	0.11	0.39	0.56	0.40	0.11	0.22	0.11	0.39	0.56	0.40

Table 5.3: Background subtraction and thresholding: Runtime in milliseconds

important factors to consider for increasing performance.

5.5.1 Closing and opening

The two operations used in closing and opening, *dilation* and *erosion*, are very similar. Both depend on their neighbouring elements, giving us some choices in how we want to implement it. We tried the following implementations, to see what gave the best runtime.

Two kernels, called multiple times

We created two kernels, to perform the dilation and erosion, respectively. They kernels are executed from the host as shown in the pseudo code in listing 5.3.

```
/* buffer1 contains the output from the thresholding step, and is used as initial input */
/* buffer2 is empty initially */
dilate_kernel(buffer1, buffer2);
erode_kernel(buffer2, buffer1);
erode_kernel(buffer1, buffer2);
dilate_kernel(buffer2, buffer1);
/* Result now in buffer1 */
```

Listing 5.3: Execution of closing and opening kernels

Global memory

We first created two simple kernels that use global memory buffers. These kernels have to perform boundary checks at the edges of the image to not read out of bounds. The *dilate* kernel is implemented as shown in 5.4. The *erode* kernel is similar, but using the *min*-function instead of *max*, as it performs the opposite operation. The values are also initialized to 255 initially.

```
__kernel void dilate_image_globalmem(__global uchar* input_img,
                                     __global uchar* output_img) {
uint width = get_global_size(0);
uint height = get_global_size(1);
uint x = get_global_id(0);
uint y = get_global_id(1);

uint id = y*width+x;

/* Initialize to 0 initially */
uchar my_val, left_val, right_val, top_val,
bottom_val, topleft_val, topright_val,
bottomleft_val, bottomright_val = 0;

/* Read my value */
my_val = input_img[id];

/*- Check boundaries for neigh. pixels -*/
if(x > 0) {
  left_val = input_img[(y*width)+(x-1)];
}
if(x < (width-1)) {
  right_val = input_img[(y*width)+(x+1)];
}
if(y > 0) {
  top_val = input_img[(y-1)*width+x];
  if(x > 0) {
    topleft_val = input_img[((y-1)*width)+(x-1)];
  }
  if(x < (width-1)) {
    topright_val = input_img[((y-1)*width)+(x+1)];
  }
}
if(y < (height-1)) {
  bottom_val = input_img[(width*(y+1))+x];
  if(x > 0) {
    bottomleft_val = input_img[((y+1)*width)+(x-1)];
  }
  if(x < (width-1)) {
    bottomright_val = input_img[((y+1)*width)+(x+1)];
  }
}

/* If any of the neighboring pixels are foreground,
the result of max will be foreground */
uchar max_top, max_center, max_bottom = 0;

max_top = max(max(topleft_val,top_val),topright_val);
max_center = max(max(left_val,my_val),right_val);
max_bottom = max(max(bottomleft_val,bottom_val),bottomright_val);

uchar max_all;
max_all = max(max(max_top,max_center),max_bottom);
```

```
55    /* Write out result */
      output_img[id] = max_all;
57  }
```

Listing 5.4: Dilate-kernel with global memory

OpenCL Image 2D

We also created two kernels using the built in Image 2D type in OpenCL. This way we can get cached reads and do not have to think about boundary checking, as reading out of bounds just return 0, without causing memory errors.

The difference in code is shown in the *dilate*-kernel in listings 5.5, 5.6, 5.7.

```
1   __kernel void dilate_image(__read_only  image2d_t input_img,
                               __write_only image2d_t output_img) {
3   /* Sampler for image */
    const sampler_t sampl = CLK_NORMALIZED_COORDS_FALSE |
5     CLK_ADDRESS_CLAMP |
      CLK_FILTER_NEAREST;
```

Listing 5.5: Dilate-kernel with Image 2D

```
    /*- Get values, don't bother checking boundaries -*/
12  uint4 my_val = read_imageui(input_img,sampl,(int2)(x, y));
    uint4 left_val = read_imageui(input_img,sampl,(int2)(x-1, y));
14  uint4 right_val = read_imageui(input_img,sampl,(int2)(x+1, y));
    uint4 top_val = read_imageui(input_img,sampl,(int2)(x, y-1));
16  uint4 bottom_val = read_imageui(input_img,sampl,(int2)(x, y+1));
    uint4 topleft_val = read_imageui(input_img,sampl,(int2)(x-1, y-1));
18  uint4 topright_val = read_imageui(input_img,sampl,(int2)(x+1, y-1));
    uint4 bottomleft_val = read_imageui(input_img,sampl,(int2)(x-1, y+1));
20  uint4 bottomright_val = read_imageui(input_img,sampl,(int2)(x+1, y+1));
```

Listing 5.6: Dilate-kernel with Image 2D

```
    /* Write out */
34  write_imageui(output_img,(int2)(x,y),max_all);
```

Listing 5.7: Dilate-kernel with Image 2D

One kernel, called once

Another possibility is to have only one kernel that performs the whole closing and opening operation. To make this work, synchronization between the work groups has to be performed between the different steps.

We created a kernel that reads the data from global into local memory before using it. This way the work items get faster access to their neighboring pixels.

Since we need synchronization between the four dilate and erode steps, we write the pixels of the work group back to global memory between each step, synchronizing the work items afterwards, and then reading back the changed border pixels into local memory before executing the next step. This way we can execute all the steps inside one kernel, avoiding the overhead of executing multiple kernels.

A negative effect of this approach is that the kernel gets more boundary checking and branching, and gets generally more complex.

Another negative impact this method has is that the reusability of the code for other purposes is limited, as we don't separate the erode and dilate functions. In our case this is not important, as trying to get the best possible runtime is our goal.

Note: This kernel had some small implementation bugs that caused some artifacts in the output, but since this shouldn't impact the performance, we include it in our runtime results for comparison. An excerpt of the code, including the first dilate step is shown in listing 5.8. *read_border_regions* is an inline OpenCL function where we check image boundaries and read back the border regions to local memory. If out of bounds, a specified value (here 0) is inserted instead.

```
__kernel void close_and_open_image(__global uchar* input_img,
                                   __global uchar* output_img,
                                   __local uchar* l_img) {
  uint width = get_global_size(0);
  uint height = get_global_size(1);
  uint block_width = get_local_size(0);
  uint block_height = get_local_size(1);
  /* 1 pixel padding around the image for borders */
  uint block_width_pad = block_width + 2;
  uint block_height_pad = block_height + 2;
  uint x = get_global_id(0);
  uint y = get_global_id(1);
  uint global_id = y*width+x;

  /* Local x and y */
  uint lx = get_local_id(0);
  uint ly = get_local_id(1);
  lx = lx + 1; // Compensate for padding
  ly = ly + 1; // of local mem
  uint local_id = (block_width_pad * ly) + lx;

  /* Read my value to local mem */
  l_img[local_id] = input_img[global_id];

  /* Read border regions (padding) to local memory */
  read_border_regions(input_img, l_img,
        width, height, x, y,
        lx, ly, block_width,
        block_height, 0);

```

```
/* Done reading to local memory. Synchronize */
barrier(CLK_LOCAL_MEM_FENCE);

/* 1st dilate */
uchar max_all = find_max_value(l_img, block_width_pad, lx, ly);
barrier(CLK_LOCAL_MEM_FENCE);
l_img[local_id] = max_all; // store value in local mem

/* Sync to global mem */
input_img[global_id] = max_all;
barrier(CLK_GLOBAL_MEM_FENCE);

/*- ---- CUT ---- -*/
/*- After this, erode, erode and dilate follows -*/
/*- Data is synced and border regions read back between each step, as above -*/
```

Listing 5.8: Closing and opening in one kernel, excerpt

Structuring element and neighbourhood

We use a structuring element of 3x3 and eight-neighbour connectivity, meaning that our pixel is in the middle of a 3x3 pixel-block, and we look at the neighbouring pixels in all 8 directions. This is the usual way to perform dilation and erosion. Other structuring elements are possible, but in our implementation we have hardcoded the use of a 3x3 structuring element, to focus on reliability and not flexibility, thus simplifying the kernel.

Experimental results

As we see in table 5.4, the Image2D kernels actually give the worst runtimes in all the cases, both compared to the global memory kernels and the single kernel with global and local memory. In some cases, the maximum runtime is more than 3 times as slow as the global memory version.

These results show us that even though OpenCL Image objects have automatic boundary checking and caching, it is not always the best choice for every implementation. Benefiting from the caching requires us to get many cache hits, and judging from the numbers, this has not been the case here. The kernels are also very simple, where the work items only read once from memory. A lesson learned from this is that implementing multiple versions of the same program can save us alot of execution time.

Comparing the global memory kernels against the single global/local-kernel, we see that the runtimes are pretty similar on the two kernels, with the min and mean runtimes of the global memory kernels being a bit lower than the single kernel. The single kernel does, however have

a slightly lower maximum runtime (around 0.10 ms) on the higher resolution video using 16x16 workgroups. The 16x16 workgroups also give the lowest runtimes in all the implementations.

For the lowest predictable max runtime, the single kernel with global and local memory will therefore probably be the best choice when using higher resolution videos, however the differences are minimal compared to the global memory two-kernel approach.

Multiple kernels with global memory												
WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
16x16	0.18	0.43	0.18	0.61	0.93	0.62	0.18	0.19	0.18	0.61	0.87	0.62
32x32	0.21	0.24	0.21	0.76	1.02	0.76	0.21	0.23	0.21	0.76	1.07	0.76
Multiple kernels with Image 2D												
16x16	0.34	0.57	0.35	1.22	1.47	1.23	0.34	0.38	0.35	1.22	1.53	1.23
32x32	0.51	0.84	0.51	1.91	2.31	1.92	0.51	0.77	0.51	1.91	2.17	1.92
One kernel with global and local memory												
16x16	0.18	0.19	0.18	0.70	0.79	0.70	0.18	0.27	0.18	0.70	0.79	0.70
32x32	0.26	0.35	0.26	1.00	1.09	1.00	0.26	0.33	0.26	1.00	1.07	1.00

Table 5.4: Closing and opening: Runtime in milliseconds

5.5.2 Connected Components Labeling

Algorithms

Our plan was to try two algorithms for CCL, finding weak spots and choosing the algorithm best suited for our problem. At the time of writing however, only one of the algorithms are functional enough for testing, so we present the runtimes of this algorithm.

Label equivalence algorithm

This algorithm is presented by Kalentev et. al. in [36]. It is a label equivalence algorithm, based on the algorithm presented in [35] by Hawick et. al. This algorithm is divided into three steps: Initialize labels, scanning, analyzation.

Initialize labels

The initialization step reads the values from the closing and opening step, and gives each pixel

a unique value, equal to its pixel position in the image. The background-pixels keep their value (0).

In this step, each pixel is independent on the others, so we launch the kernel as a 2D-grid the size of the image. We try 16x16, 32x32 and automatic work group sizes. The input and output is Image2D in our implementation, but we could just as well used global memory here, since this step does not depend on neighboring pixels.

Scanning

In the scanning step, each work item look at its neighboring pixels and set its value to the lowest non-zero value of the neighbors and itself. The scanning phase iterates multiple times, until the neighbors have the same label (or 0 if background), and then terminates execution. We have used Image2D as input to this kernel, to get cached reads when accessing neighboring pixels. What could also have been tested was to use global memory instead, and read the pixels into local memory. As we planned to get the other algorithm up and running, we haven't tested this.

We did, however try to improve execution a bit by using 8 neighboring pixels in the comparison above, instead of only 4. This way, more pixels are compared each iteration (even though more elements has to be read). The results of this can be seen below, but we have only tested it on a 16x16 work group size.

Analyzation

In the analyzation step, the final merging of the labels take place, where each work item run in a loop. Using its label as lookup position in memory, it checks if the label at that position is equal to its own label. If not, the new label is taken, and this is again used as a reference. This goes on until the current label and the label at global_mem[label] is the same number. We use Image2D for reading the initial labels in this step, and global memory for the rest, to be able to update the labels in a loop, as Image2D is read only or write only within a kernel.

Experimental results

Table 5.5 shows the runtime of the implemented CCL algorithm. As we can see, the max runtime on both small and especially on large videos is very high.

We note that the work group size has a pretty noticable impact on performance, especially on the large videos. The 16x16 work group is definitely the fastest. We also notice that the runtime difference between the two videos is pretty big, giving the highest runtime on *Video 1*. As the foreground object in *Video 1* is closer to the camera than in *Video 2*, the objects will be

bigger, and finding all the connected pixels of this object will then demand more iterations of the scanning kernel than in *Video 2*.

Using 8 neighbors instead of 4 for the scanning part of the algorithm was only tested for a 16x16 work group size. As our experiments show, the runtime is considerably lower with this approach. As the scanning part now does more in each iteration, it has to run fewer iterations, and this may explain the shorter total runtime. We run around 40 ms faster on the larger videos with this approach. The mean and min runtimes are a bit higher, but on this step it is the worst case that is most important.

Notes

As the chosen algorithm is naturally hard to parallelize, accessing global values over the whole image, we started the implementation of another, more parallel algorithm. The algorithm is presented by Stava and Benes in the book GPU Computing Gems Emerald Edition [37]. It is more sophisticated than the abovementioned algorithm, containing multiple merging steps and utilization of local memory, for better performance.

As this implementation was not finished in time, we can not give any runtime results, but we believe that it could give a good performance increase on the CCL-step, as it utilizes the GPU much better than the aforementioned algorithm.

WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
4 neighbor lookup in scanning part												
Auto	0.22	52.40	7.72	1.03	254.81	47.09	0.33	30.70	2.83	1.97	190.12	71.90
16x16	0.20	49.91	6.91	0.82	188.04	33.40	0.31	33.76	2.49	1.81	137.98	49.99
32x32	0.25	55.79	7.71	1.03	233.95	40.91	0.36	37.61	2.85	2.25	174.79	65.41
8 neighbor lookup in scanning part												
16x16	0.22	36.43	7.96	0.97	142.80	34.12	0.33	34.07	3.21	1.70	95.99	38.51

Table 5.5: Connected component labeling: Runtime in milliseconds

5.6 Profile extraction

The profile extraction has not been implemented on GPU. However, we have some thoughts to how an implementation could be approached.

If there are multiple silhouettes detected in the image, the profile extraction has to run one time for each of these silhouettes, covering it from each of the sides. Keeping different parts of the GPU busy with different silhouettes at the same time to maximize utilization is one thing to think about.

Since this step also requires accessing neighboring pixels, reading in data to local memory before using it would be a good idea, to save global memory reads and maximize performance.

5.7 Background update

5.7.1 Limitations

In the background update step, we have omitted the part about changes in background geometry, and only implemented the background update for gradual changes in lighting. As this is a very simple kernel, we have not tried any specific optimizations here. We have executed it with different work group sizes.

5.7.2 Experimental results

The runtimes are found in table 5.6. As we can see, the 16x16 work group performs best of the three choices.

WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
Auto	0.14	0.18	0.15	0.63	0.69	0.64	0.14	0.18	0.15	0.63	0.70	0.63
16x16	0.13	0.22	0.13	0.48	0.57	0.49	0.13	0.15	0.13	0.48	0.55	0.49
32x32	0.17	0.23	0.17	0.62	0.71	0.63	0.17	0.26	0.17	0.62	0.64	0.62

Table 5.6: Background update: Runtime in milliseconds

5.8 Total runtime of the algorithm

To get a sense of how fast our entire implementation is, we have performed a total runtime comparison of all the steps above. We use the configurations that gave best runtimes in the experiments above on each step, to get the best runtimes in total. As we already know that the CCL step is the slowest part, we have tested with and without this step. This does not influence

the video, as the CCL step only finds the connected components in the image, but we do no filtering based on this in our implementation.

5.8.1 Experimental results

Table 5.7 shows our runtime results. On the small videos, we get around 38 ms total runtime in the worst case, giving us around 26 frames per second. On the large videos, our max runtime is 148.49 ms, giving a frame rate of around 6,7 fps. The min and mean runtimes on the large videos are much lower, between 6-8 ms min runtime and 40-50 ms mean, showing that the runtime will vary very much depending on the foreground regions in the image.

Without the CCL step, we get some quite different results. In the worst case, on the large videos, we get a runtime of 5.72 ms, or more than 170 frames per second. It shows both how fast the rest of the implementation is, and how much of the runtime is used by the CCL step.

WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
With Connected Component Labeling (CCL)												
N/A	1.61	37.91	9.42	6.44	148.49	39.74	1.99	35.56	4.66	7.16	101.69	44.13
Without Connected Component Labeling (CCL)												
N/A	1.36	1.75	1.38	5.20	5.64	5.23	1.36	1.85	1.38	5.22	5.72	5.24

Table 5.7: Full algorithm on GPU: Runtime in milliseconds

5.9 Summary

In this chapter we have presented our GPU implementation of the silhouette extraction algorithm presented in chapter 2. We have explored different optimizations, and shown runtime results on different video. In chapter 6 we compare this GPU implementation to our CPU-implementation, looking at differences in execution time in each of the steps, and in total.

Chapter 6

Comparison between CPU and GPU implementation

In this chapter we look at the difference in runtime between our GPU implementation described in chapter 5 and our CPU-implementation of the same algorithm.

6.1 Introduction

To be able to measure the performance improvements of our GPU implementation, a reference CPU-implementation was needed. We implemented the same pipeline on CPU as on GPU. Instead of implementing every step of the algorithm ourselves, we have used already existing algorithms from the **Open Computer Vision library (OpenCV)** [39] for some of the processing.

For the most accurate results, the same training data has been used for both CPU and GPU implementations, as well as the parameters and thresholds to the algorithms. The performance measurements are also performed the same way as on GPU, calculating min, max and mean of the runtimes.

The following sections describe the differences between GPU and CPU runtime in each of the different steps. We use the GPU runtimes with the shortest max time from chapter 5 to compare against the CPU implementation A runtime comparison of the whole pipeline will also be shown.

6.1.1 Notes about CPU performance

The CPU implementation is compiled with the *-O3* flag, but no other optimization approaches have been followed. A tuned and optimized CPU version would probably give better runtimes

than those we have measured, but since we have focused on the GPU implementation in this thesis, optimizing the CPU version was out of scope.

The tests have been run on a machine with an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz processor and 8 GB of memory. This is the same machine the GPU implementation is tested on.

6.2 Training step

The training step is pretty straight forward, doing per-pixel calculations in a loop. The same running statistics approach as mentioned in chapter 5 has been implemented here.

6.2.1 Experimental results

As we see in table 6.1, the performance increase in our GPU version is very high on small and large videos, giving maximum values a runtime that is around 14 times faster on the small videos, and up to 20 times faster on the large videos. Even though the training step need to copy a certain amount of data to the GPU, the performance improvements are very high. As this step is per-pixel parallel, this was expected, as memory reads and writes can be performed coalesced and there is no access of neighboring pixels that can slow down the parallel execution.

CPU Implementation												
WG	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
N/A	7.07	10.98	7.39	28.55	44.32	29.89	7.06	10.98	7.30	28.61	44.36	29.46
GPU Implementation												
16x16	0.44	0.77	0.46	1.45	2.22	1.53	0.44	0.77	0.46	1.44	2.19	1.50

Table 6.1: Training step - CPU and GPU - runtime in milliseconds

6.3 Background modeling

The background modeling contains usage of quite a bit of math operations like pow, exp, sqrt and gamma functions. On the CPU these functions can be pretty slow, and running them sequentially will take some time.

6.3.1 Experimental results

This may be one of the explanations to the high runtime on the CPU shown in table 6.2.

As the GPU is optimized for these kinds of operations, this, combined with the power of parallelization show that the GPU version runs more than 100 times faster in all cases. An optimized CPU version would probably weigh up for some of this, but it is still a very high increase.

CPU Implementation												
WG	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
N/A	73.38	75.32	73.74	287.97	294.69	288.94	73.38	75.11	73.97	291.19	295.63	292.82
GPU Implementation												
16x16	0.60	0.67	0.61	2.35	2.44	2.36	0.60	0.70	0.61	2.36	2.43	2.36

Table 6.2: Background modeling - CPU and GPU - runtime in milliseconds

6.4 Background subtraction, thresholding and foreground extraction

6.4.1 Subtraction and thresholding

The background subtraction and thresholding is performed by looping over all the pixels, extracting the background and performing thresholding.

6.4.2 Experimental results

The data in table 6.3, shows pretty clear that this is a per-pixel operation, giving the GPU a big advantage of executing in parallel. The step does contain some if-statements, so some branching will create a bit of serial execution on the GPU. We do however still get a speedup of around 13x on the max times of the large videos. The smaller videos have a little lower speedup, of around 7-8x faster. The speedups on min and mean execution time is around 14-18x.

CPU Implementation												
WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
N/A	1.10	1.54	1.21	4.52	5.76	5.04	1.12	1.46	1.16	4.78	6.03	5.00
GPU Implementation												
16x16	0.08	0.19	0.08	0.28	0.43	0.28	0.08	0.19	0.08	0.28	0.45	0.28

Table 6.3: Background subtraction and thresholding - CPU and GPU - runtime in milliseconds

6.5 Morphological processes

6.5.1 Closing and opening

Implementation

The closing and opening is performed using the *cv::morphologyEx* function from OpenCV. This performs the dilation and erosion the same way as described in the previous chapters, with a 3x3 kernel.

Experimental results

Table 6.4 shows the runtime results of the closing and opening. Even though the runtime on CPU for the smaller videos seem low, we can see that the GPU version is around 14-15 times faster on all videos and sizes.

CPU Implementation												
WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
N/A	2.69	3.02	2.71	10.65	11.83	10.88	2.69	3.01	2.73	10.69	11.14	10.81
GPU Implementation												
16x16	0.18	0.19	0.18	0.70	0.79	0.70	0.18	0.27	0.18	0.70	0.79	0.70

Table 6.4: Closing and opening - CPU and GPU - runtime in milliseconds

6.5.2 Connected Component Labeling

The connected component labeling step is where we hit a weak spot in our GPU implementation. As we haven't implemented very efficient algorithm on GPU, the CPU version is actually faster in this case.

Experimental results

Table 6.5 shows the runtime results, and as we can see, the CPU version have between 1.2 and 1.5 times faster runtimes on Video 1 and the high resolution Video 2. The low resolution Video 2 is more than 8 times faster on CPU, in the worst case. This shows us that a more parallel and effective CCL algorithm is definitely needed on GPU.

CPU Implementation												
WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
N/A	0.29	27.15	0.87	1.17	127.07	12.14	0.30	3.91	0.51	1.78	66.79	15.89
GPU Implementation												
16x16	0.22	36.43	7.96	0.97	142.80	34.12	0.33	34.07	3.21	1.70	95.99	38.51

Table 6.5: Connected Component Labeling - CPU and GPU - runtime in milliseconds

6.6 Background update

The background update is very simple, updating the mean and variance running average for the background pixels.

6.6.1 Experimental results

Table 6.6 shows that the GPU version in the worst case (max runtime) is between 8 and 12 times faster than the CPU version. This is expected, as there is almost no branching and a very simple kernel, making it easy to run in parallel.

CPU Implementation												
WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
N/A	1.26	1.78	1.28	5.17	6.41	5.29	1.27	1.73	1.30	5.18	6.06	5.25
GPU Implementation												
16x16	0.13	0.22	0.13	0.48	0.57	0.49	0.13	0.15	0.13	0.48	0.55	0.49

Table 6.6: Background update - CPU and GPU - runtime in milliseconds

6.7 Measurement of the whole pipeline

A total comparison of the whole pipeline on CPU and GPU shows us how much faster our GPU implementation is.

6.7.1 Experimental results

In table 6.7, we see that max runtimes on GPU are between 2 and 3.4 times lower and on CPU. The highest runtime decrease is seen on the large video 2, running 3.4 times faster on GPU. Considering the CCL step is actually slower on GPU, this is a quite high runtime improvement.

WG size	**Video 1**						**Video 2**					
	512x288			1024x576			512x288			1024x576		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
CPU Implementation												
N/A	69.83	96.27	70.57	277.00	410.70	289.46	69.84	73.78	70.30	278.65	347.30	294.26
GPU Implementation												
N/A	1.61	37.91	9.42	6.44	148.49	39.74	1.99	35.56	4.66	7.16	101.69	44.13

Table 6.7: Full algorithm - CPU and GPU - runtime in milliseconds

6.8 Summary

In this chapter we have compared the runtime of our GPU and CPU implementations. The GPU version runs between 2 and 3.4 times faster in total, with only the CCL step giving an increased runtime on GPU compared to CPU. In the next chapter we look at the quality of the silhouettes produced by our implementation.

Chapter 7

Quality assesment of the silhouette extraction algorithm

In this chapter we look at the quality of the silhouettes produced by our silhouette extraction algorithm.

7.1 Introduction

We perform a brief evaluation of the quality of the silhouettes produced by the silhouette extraction algorithm we have implemented. We count pixels wrongly detected as either background or foreground in a small set of sample images.

7.2 Ground truth

To be able to measure how accurate the detected silhouettes are, we have created ground truth data for 10 frames in each of our two videos. The frames have been selected by hand, to make sure we have different results, not only the frames where we get the best results.

The ground truth is created by editing the original images by hand and drawing around the foreground objects in them. The foreground is then masked as white, and the background as black.

7.3 Measurements

7.3.1 Notes and limitations

A comparative and extensive quality assessment would require us to implement multiple known approaches for silhouette extraction, and compare our results to those, giving us a measure to how well our chosen algorithm performs compared to others. As our main focus has been to implement one algorithm, and to do so on GPU for high performance, this is out of scope for this thesis.

Instead, we measure our own implementation against ground truth data, giving us measures of how accurate our chosen algorithm is at the current state of implementation.

CPU implementation

In our CPU implementation we have added the step of filtering out regions in the image that are smaller than a given threshold, as presented in chapter 4. Note that this step is not included in our performance testing. As this is not included in our current GPU implementation, we use the CPU implementation with this enabled for creating the test data, giving a more complete representation of the algorithm used. The profile extraction step is, however, not implemented, so foreground objects are expected to have false negative errors, as described below.

Video size

We have only performed these measurements on video with size 1024x576.

7.3.2 False positive and false negative errors

For each of our two test-videos, we compare 10 selected frames, where a foreground object is present. Each frame is compared to its corresponding ground truth data.

We count two kinds of errors: **False positive (FP)** and **false negative (FN)**. FP errors are pixels that are detected as foreground where they should be background. FN on the other hand are pixels detected as background when they should be foreground. The error percentages are calculated with equation 7.1.

$$\text{error} = \frac{\text{number of erroneous pixels}}{\text{number of foreground pixels in ground truth}} x100(\%) \qquad (7.1)$$

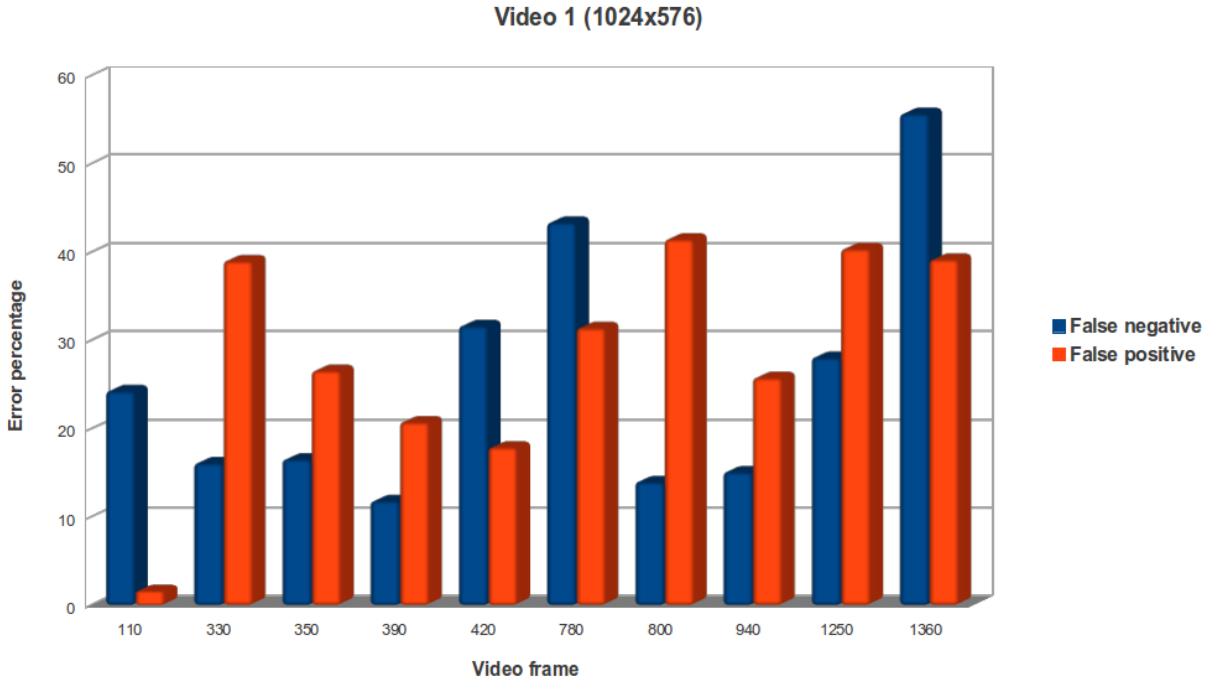


Figure 7.1: Video 1 - False negative and false positive errors

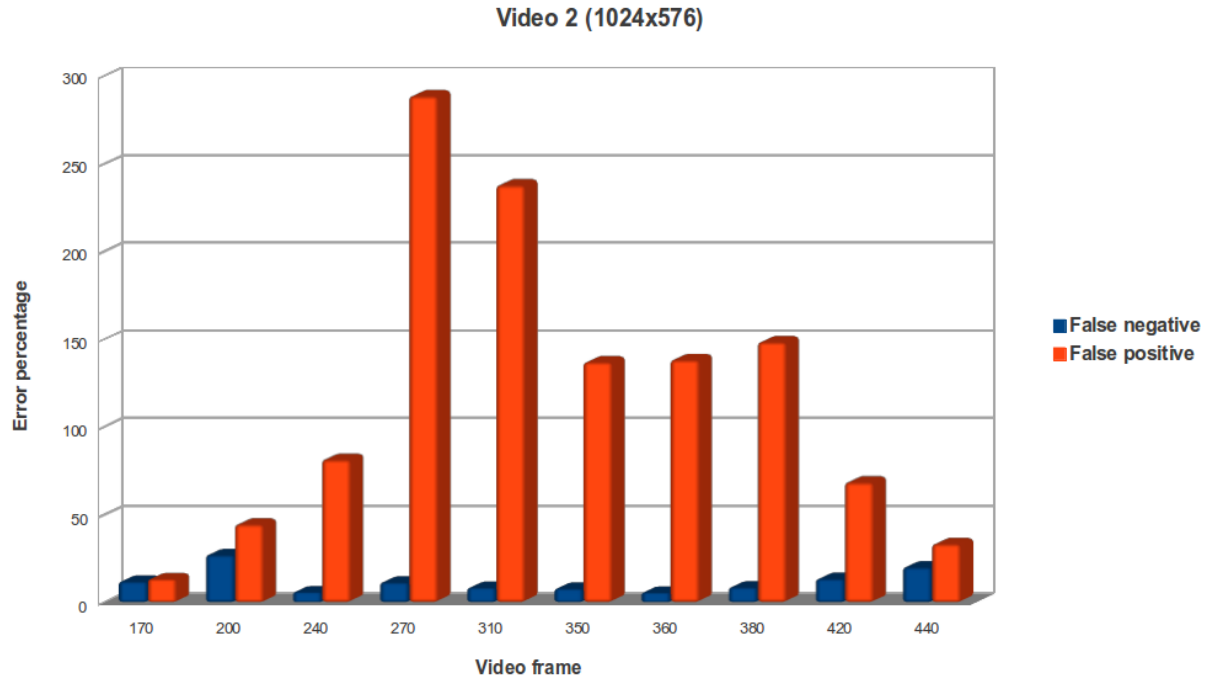


Figure 7.2: Video 2 - False negative and false positive errors

7.3.3 Experimental results

In figure 7.1 and 7.2 we show the FN and FP errors on different frames of the two videos.

Video 1

In *Video 1*, we can see that the FP errors in some cases are quite high, the highest being 41.45%, in frame 800. The reason for this can be lights, reflections and shadows being incorrectly identified as foreground. Original frame, ground truth and extracted foreground of frame 800 is shown in figure 7.3. The FN errors are in many cases quite low, but in some frames they are very high, e.g. 55.66% in frame 1360 and 43.36% in frame 780.

The effect of this can be seen in figure 7.4, where the foreground silhouette is full of holes. This frame also has 31.38% FP errors, where some of it is clearly caused by reflection in the glass frame on the wall.

In some cases, refinement of the detected foreground, and increasing the threshold on how large foreground objects can be filtered out might help. The main problem, though, is setting and adapting the thresholds correctly, so shadow regions are recognized as suspicious foreground, and then filtered out with the shadow removal step. Since we have just hand-tuned thresholds and found something that works OK, we don't get the best results possible.

Frame 350 and 390 have quite low FP and FN errors, the results compared to ground truth can be seen in figure 7.5.

Video 2

In *Video 2*, the results are quite different, where the highest FN error, in frame 200, is at 27.21%. The FP errors are, however, very high. In frame 270, the FP error percentage is 288.68%, meaning that alot more than the foreground object has been detected as foreground. As we can see in figure 7.6, light in the roof and reflections in the glass window in the top of the scene is wrongly detected as foreground objects, creating the high error rate. Better tuning of the thresholding parameters might help with this. As the FN errors are so low compared to *Video 1*, we have probably tuned it to be way too permissive in regard to what ends up in the *reliable foreground* category. We have not used very much time on tuning the parameters.

Frames 170, 200 and 440 have the lowest FP error rates in video 2. They are shown in figure 7.7.

Discussion

As we can see, the extracted silhouettes are far from perfect, in some cases giving lots of false positive errors, and other cases quite high false negative error rates. Some errors are expected, as we haven't tuned the parameters for thresholding very much, and as we can see in some cases, reflections and shadows can cause false positive errors. False negative errors can cause

holes inside a detected foreground object. Implementing the profile extraction step, described in chapter 4, could help improve the silhouettes significantly.

Automatic thresholding

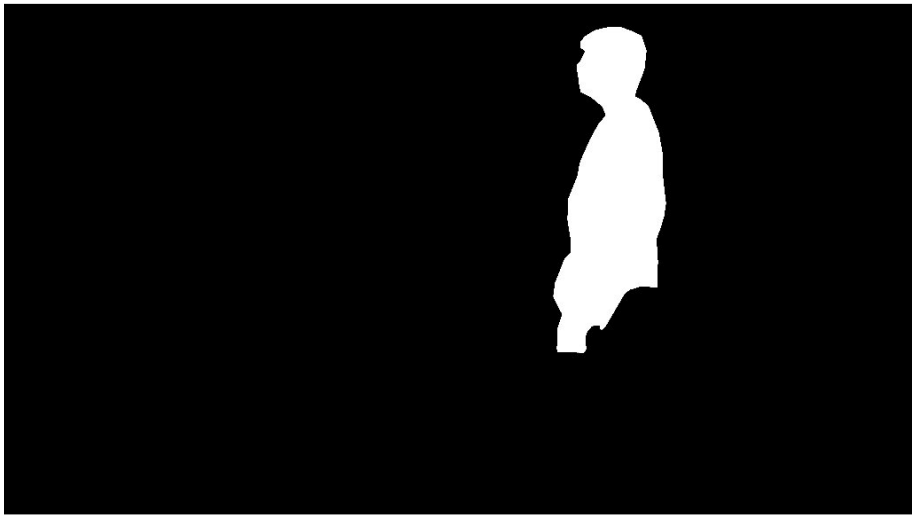
As long as the thresholding step requires very manual tuning of the parameters, this algorithm is pretty hard to use. The thresholds are selected manually, and have to be tuned for new environments - a time consuming and inaccurate process. For a more adaptable algorithm, a more automated way to choose thresholds should be considered. In [40], Otsu presented a nonparametric and unsupervised method to threshold gray level images automatically. Sun et. al. [41] presented an automatic thresholding algorithm for moving foreground objects, presenting improvements to the method presented by Otsu [40]. Applying a scheme like this after the background subtraction in our algorithm, instead of manually setting the thresholds could be worth looking into for better results, and a much more adaptive algorithm.

7.4 Summary

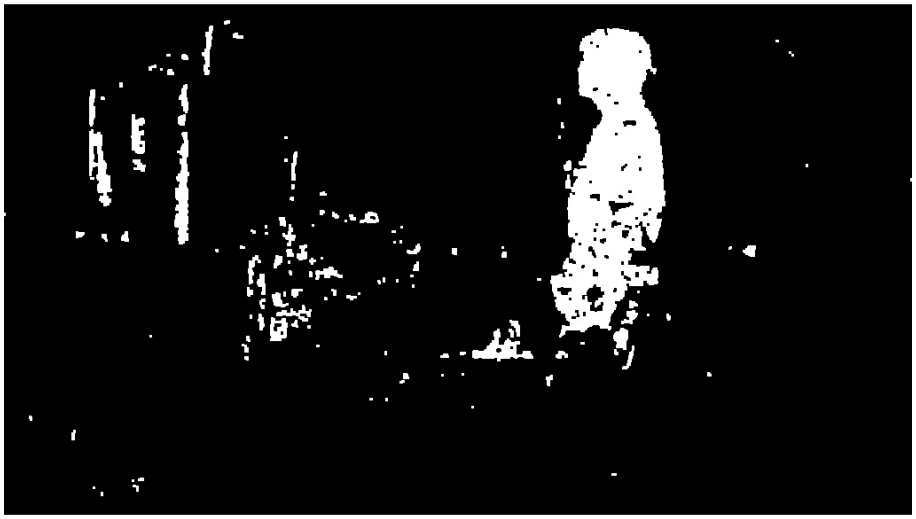
In this chapter we have looked at the quality of the silhouettes generated by our algorithm. We have measured false negative and false positive errors in selected frames and discussed our findings. We have also presented some possible improvements, to get better results.

In the next chapter we conclude the thesis and present future work.

Video 1 - Frame 800 - Original frame



Video 1 - Frame 800 - Ground truth



Video 1 - Frame 800 - Extracted foreground

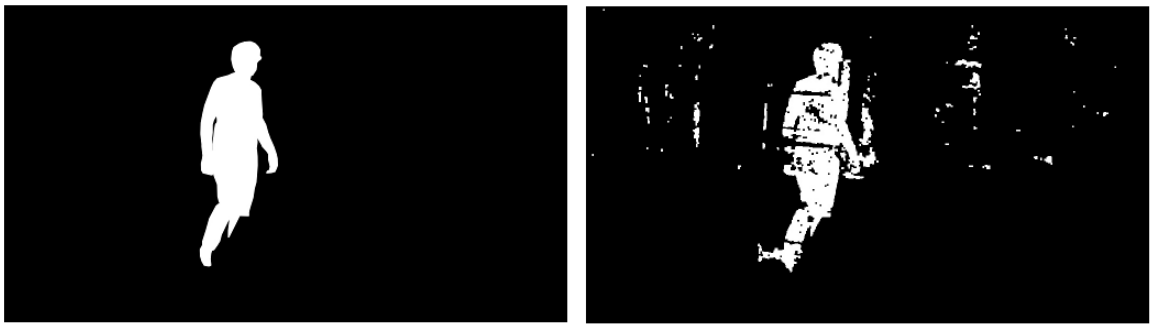
Figure 7.3: Video 1 - Frame 800 - False positive errors

Video 1 - Frame 780 - Original frame

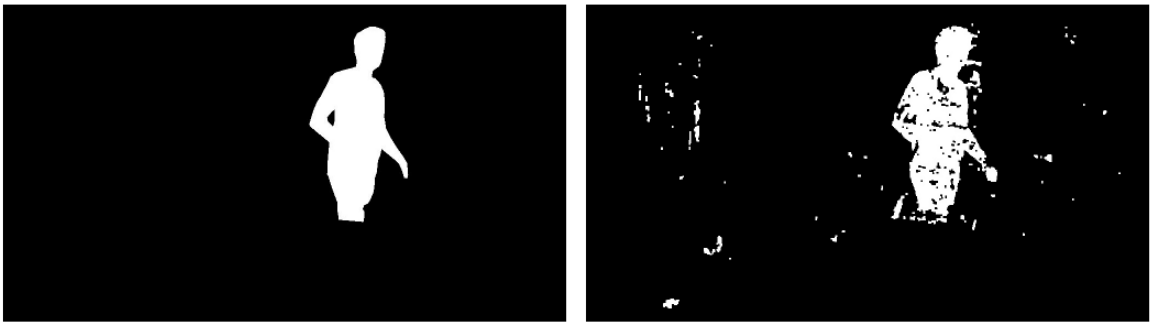
Video 1 - Frame 780 - Ground truth

Video 1 - Frame 780 - Extracted foreground

Figure 7.4: Video 1 - Frame 780 - False negative and false positive errors



Frame 350 - Ground truth and extracted foreground



Frame 390 - Ground truth and extracted foreground

Figure 7.5: Video 1 - Frame 350 and 390 - Low FP and FN

Video 2 - Frame 270 - Original frame

Video 2 - Frame 270 - Ground truth

Video 2 - Frame 270 - Extracted foreground

Figure 7.6: Video 2 - Frame 270 - 288.68% FP error

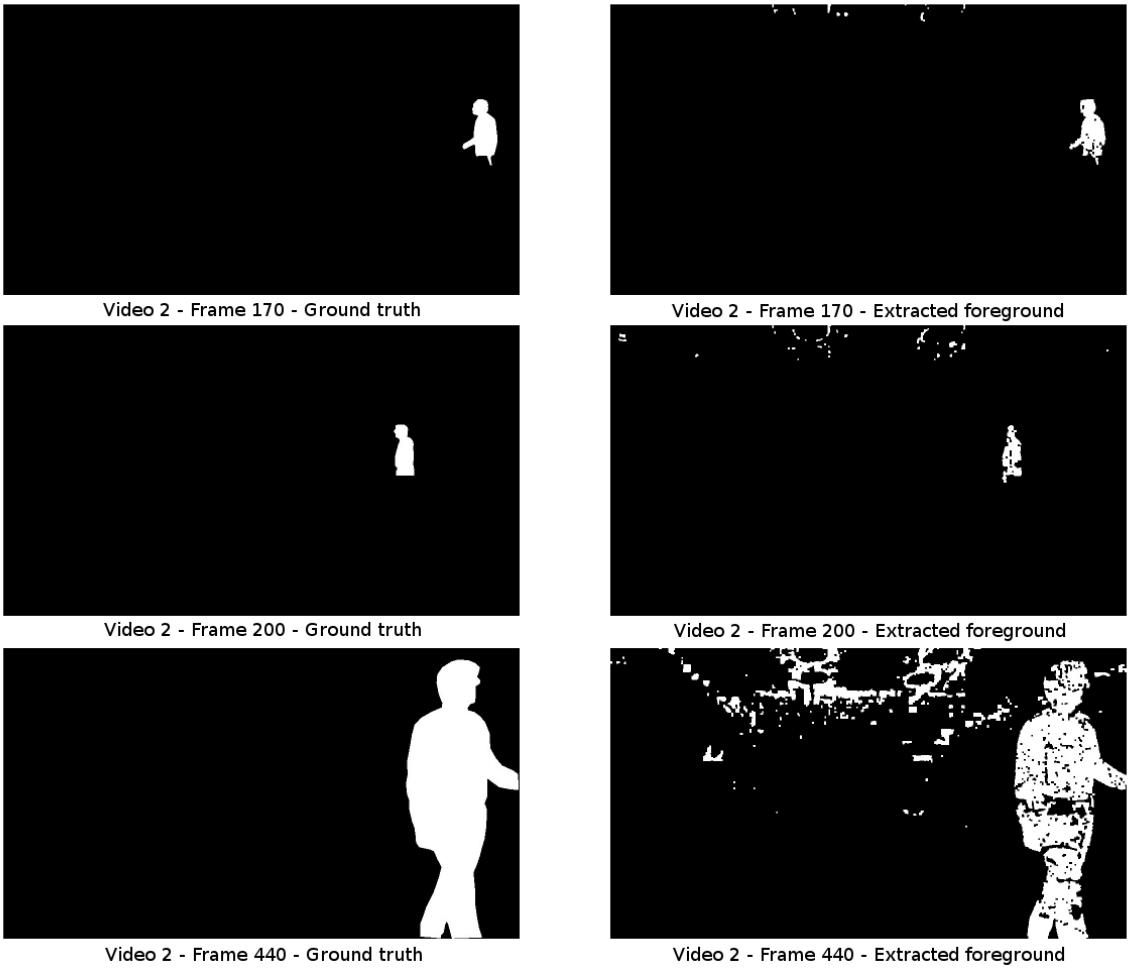


Figure 7.7: Video 2 - Frame 170, 200 and 440 - low error rates

Chapter 8

Conclusion

In this thesis we have presented a silhouette extraction algorithm, and investigated how to implement it on GPU. We have implemented most parts of the algorithm on GPU, and on CPU as a reference. Our experimental results show that the performance increase in most steps is very high compared to our CPU implementation. The connected component labeling step does however need a faster algorithm/implementation, as this part is even slower on GPU than on CPU in the worst cases. We have mentioned an altenative algorithm for this that should possibly improve performance.

Our implementation is a good starting point for achieving real time silhouette extraction that can be used in multiple computer vision applications.

The quality of the silhouettes produced by our implementation are fairly good, but because of shadow regions and reflections being wrongly detected as foreground, we get a lot of false positive errors in some cases, detecting elements that is background as foreground. In other cases, we get a high number of false negative errors, resulting in holes inside the silhouettes and missing parts. As mentioned in chapter 7, tuning the thresholds or implementing an automatic thresholding scheme could help with these problems. Because we omited the profile extraction step of the algorithm in our implementation, there are also holes inside the foreground objects at times.

8.1 Future work

The GPU implementation presented in this thesis is a good starting point for a real time high quality silhouette extraction algorithm. There are, however several open issues.

8.1.1 Faster connected component labeling

As we mentioned in our results and the conclusion, a faster connected components labeling algorithm is needed to get full real-time performance. An algorithm that might work well is mentioned in chapter 5.

8.1.2 Implementation of profile extraction

An implementation of the profile extraction described in chapter 4 would refine the silhouettes even more, giving a better total result, covering holes inside foreground objects.

8.1.3 Improvement of thresholding

As we have mentioned, the thresholding step in the algorithm is dependent on manually set threshold values, specifically tuned for certain environments. This is not very portable, making the algorithm hard to adapt to new environments. Implementing an automatic, adaptive thresholding scheme would therefore be worth looking into. Possible algorithms for doing this is mentioned in chapter 7.

Bibliography

[1] S. Pierard, A. Lejeune, and M. Van Droogenbroeck. A probabilistic pixel-based approach to detect humans in video streams. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 921 –924, may 2011.

[2] Jordi Salvador, Xavier Suau, and Josep R. Casas. From silhouettes to 3d points to mesh: towards free viewpoint video. In *Proceedings of the 1st international workshop on 3D video processing*, 3DVP '10, pages 19–24, New York, NY, USA, 2010. ACM.

[3] Simon Prince, Adrian David Cheok, Farzam Farbiz, Todd Williamson, Nik Johnson, Mark Billinghurst, and Hirokazu Kato. 3d live: Real time captured content for mixed reality. In *Proceedings of the 1st International Symposium on Mixed and Augmented Reality*, ISMAR '02, pages 7–, Washington, DC, USA, 2002. IEEE Computer Society.

[4] *NVIDIA CUDA C Programming Guide version 4.2*, 2012. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (2012-05-01).

[5] OpenCL - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/ (2012-04-07).

[6] Alan M. Mcivor. Background Subtraction Techniques. 2000.

[7] M. Piccardi. Background subtraction techniques: a review. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 4, pages 3099 – 3104 vol.4, oct. 2004.

[8] Y. Benezeth, P.M. Jodoin, B. Emile, H. Laurent, and C. Rosenberger. Review and evaluation of commonly-implemented background subtraction algorithms. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1 –4, dec. 2008.

[9] M. Hedayati, W.M.D.W. Zaki, and A. Hussain. Real-time background subtraction for video surveillance: From research to reality. In *Signal Processing and Its Applications (CSPA), 2010 6th International Colloquium on*, pages 1 –6, may 2010.

[10] R. Cucchiara, C. Grana, M. Piccardi, and A. Prati. Detecting moving objects, ghosts, and shadows in video streams. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(10):1337 – 1342, oct. 2003.

[11] N. J. B. McFarlane and C. P. Schofield. Segmentation and tracking of piglets in images. *Machine Vision and Applications*, 8:187–193, 1995. 10.1007/BF01215814.

[12] N. Friedman and Russell. Image segmentation in video sequences: A probabilistic approach. In *Thirteenth Conf. on Uncertainty in Artificial Intelligence*, pages 175–181, 1997.

[13] C. Wren, A. Azarbayejani, T. Darrell, and A. Pentland. Pfinder: real-time tracking of the human body. In *Automatic Face and Gesture Recognition, 1996., Proceedings of the Second International Conference on*, pages 51 –56, oct 1996.

[14] C. Stauffer and W.E.L. Grimson. Adaptive background mixture models for real-time tracking. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 2, pages 2 vol. (xxiii+637+663), 1999.

[15] A. Shimada, D. Arita, and R. Taniguchi. Dynamic control of adaptive mixture-of-gaussians background model. In *Video and Signal Based Surveillance, 2006. AVSS '06. IEEE International Conference on*, page 5, nov. 2006.

[16] Ahmed Elgammal, David Harwood, and Larry Davis. Non-parametric model for background subtraction. In *FRAME-RATE WORKSHOP, IEEE*, pages 751–767, 2000.

[17] Shengping Zhang, Hongxun Yao, and Shaohui Liu. Spatial-temporal nonparametric background subtraction in dynamic scenes. In *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*, pages 518 –521, 28 2009-july 3 2009.

[18] P. Kumar, K. Sengupta, and S. Ranganath. Real time detection and recognition of human profiles using inexpensive desktop cameras. In *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, volume 1, pages 1096 –1099 vol.1, 2000.

[19] S. Jabri, Z. Duric, H. Wechsler, and A. Rosenfeld. Detection and location of people in video images using adaptive fusion of color and edge information. In *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, volume 4, pages 627 –630 vol.4, 2000.

[20] Mao-Hsiung Hung, Jeng-Shyang Pan, and Chaur-Heh Hsieh. Speed up temporal median filter for background subtraction. In *Pervasive Computing Signal Processing and Applications (PCSPA), 2010 First International Conference on*, pages 297 –300, sept. 2010.

[21] P. Rosin. Thresholding for change detection. In *Computer Vision, 1998. Sixth International Conference on*, pages 274 –279, jan 1998.

[22] M. Sezgin and B. Sankur. Survey over image thresholding techniques and quantitative performance evaluation, 2004.

[23] D. Chen, S. Denman, and C. Fookes. Accurate silhouette segmentation using motion detection and graph cuts. In *Information Sciences Signal Processing and their Applications (ISSPA), 2010 10th International Conference on*, pages 81 –84, may 2010.

[24] Alexandre Alahi, Luigi Bagnato, Damien Matti, and Pierre Vandergheynst. Foreground Silhouettes Extraction robust to Sudden Changes of background Appearance. In *IEEE International conference on Image Processing*, 2012.

[25] *NVIDIA - "OpenCL Programming Guide for the CUDA Architecture"*, 2012. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf (2012-04-13).

[26] *The OpenCL Specification*, 2011. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf (2012-04-13).

[27] Hansung Kim, Ryuuki Sakamoto, Itaru Kitahara, Tomoji Toriyama, and Kiyoshi Kogure. Robust foreground extraction technique using gaussian family model and multiple thresholds. In Yasushi Yagi, Sing Kang, In Kweon, and Hongbin Zha, editors, *Computer Vision – ACCV 2007*, volume 4843 of *Lecture Notes in Computer Science*, pages 758–768. Springer Berlin / Heidelberg, 2007.

[28] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):pp. 419–420, 1962.

[29] Sandia Report. Formulas for robust , one-pass parallel computation of covariances and arbitrary-order statistical moments. *Contract*, SAND2008-6(September):1–18, 2008.

[30] YUV - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/YUV (2012-04-30).

[31] Azriel Rosenfeld and John L. Pfaltz. Sequential operations in digital picture processing. *J. ACM*, 13(4):471–494, October 1966.

[32] Michael B. Dillencourt, Hannan Samet, and Markku Tamminen. A general approach to connected-component labeling for arbitrary image representations. *J. ACM*, 39(2):253–280, April 1992.

[33] Hanan Samet. Connected component labeling using quadtrees. *J. ACM*, 28(3):487–501, July 1981.

[34] Roshan Dharshana Yapa and Harada Koichi. A connected component labeling algorithm for grayscale images and application of the algorithm on mammograms. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 146–152, New York, NY, USA, 2007. ACM.

[35] K.A. Hawick, A. Leist, and D.P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Computing*, 36(12):655 – 678, 2010.

[36] Oleksandr Kalentev, Abha Rai, Stefan Kemnitz, and Ralf Schneider. Connected component labeling on a 2d grid using cuda. *Journal of Parallel and Distributed Computing*, 71(4):615 – 620, 2011.

[37] Wen-mei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[38] George Stockman and Linda G. Shapiro. *Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[39] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[40] A threshold selection method from gray-level histograms. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(1):62 –66, jan. 1979.

[41] Zhihai Sun, Shan an Zhu, and Dawei Zhang. Real-time and automatic segmentation technique for multiple moving objects in video sequence. In *Control and Automation, 2007. ICCA 2007. IEEE International Conference on*, pages 825 –829, 30 2007-june 1 2007.