universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master‘s Thesis

Real-Time Facial Feature Detection on Mobile Graphics Processing Units

verfasst von / submitted by

Helf Christopher, MSc. BSc.

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna 2018

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von / Supervisor:	Univ. Prof. Dr. Helmut Hlavacs

Abstract

Facial feature detection is a key instrument in enhancing several consumer and enterprise-grade mobile apps. Even though recent advances have vastly improved the speed and accuracy of detection of facial feature points, there is a great lack of implementations available specifically designed for mobile graphics processing units (GPU). This thesis describes the implementation of an open-source facial feature detection pipeline residing entirely on the GPU written for the iOS platform achieving real-time performance including a number of algorithms key to computer vision problems, such as face detection using haar classifiers, regression trees or optical flow. We show that by encoding a single set of GPU-instructions per frame, performance can be vastly improved as compared to naive CPU or even GPU implementations without compromising accuracy. By open-sourcing this implementation, we aim at facilitating the creation of apps relying on performant facial feature detection in both the academic and industrial domain.

Zusammenfassung

Die Erkennung von Gesichtsmerkmalen oder Facial Feature Detection ist ein Schlüsselinstrument bei der Verbesserung von mobilen Apps für Endanwender und Unternehmen. Obwohl signifikante Fortschritte in der Literatur die Geschwindigkeit und Genauigkeit der Erkennung von Gesichtsmerkmalen erheblich verbessert haben, gibt es einen großen Mangel an Implementierungen, die speziell für mobile Grafikprozessoren (GPU) entwickelt werden. Diese Masterarbeit beschreibt die Implementierung einer Open-Source Gesichtsfeature-Erkennungs-Pipeline, die vollständig für GPU Processing auf der iOS-Platform geschrieben wurde und dabei Echtzeit-Performance erreicht. Dies wird erzielt durch Redesign und Optimierung einiger Algorithmen aus dem Computer Vision Bereich wie etwa Haar-Cascade Face Detection, kaskadierte Regressionsbäume für Gesichtsmerkmal Detektierung oder Optical Flow. Wir zeigen, dass durch die Kodierung eines einzigen Satzes von GPU-Anweisungen pro Frame die Leistung im Vergleich zu naiven CPU- oder sogar GPU-Implementierungen erheblich verbessert werden kann, ohne dabei Genauigkeit zu beeinträchtigen. Durch die Open-Source-Implementierung dieser Arbeit möchten wir die Erstellung von Apps erleichtern, die auf einer performanten Erkennung von Gesichtsmerkmalen im akademischen und industriellen Bereich basieren.

Contents

List of Figures

List of Tables

List of Algorithms

List of Codes

1 Introduction

Smart phones have become the most important technology used on a daily basis in our recent lives and are developing at an incredible rate with regards to computing power. Compute intensive tasks from computer vision and machine learning, once only processable on desktop computers, are slowly but steadily being employed to devices in our pockets. This allows numerous consumer and industrial apps to include modern image and video-based features into their core functionality. One of the key developments in this area are graphics processing units or GPUs, which have enabled developers to make use of high levels of data parallelism and computation for their tasks. Once having been designed for specific problem settings in the past, these chips have long moved on to be fully programmable for every day tasks that are suitable to their architecture and parallelism. Computer vision as a well-researched area in general due to its many applications for human computer interaction has quickly embraced the changes in this technology and achieved impressive results with regards to e.g. accuracy of detecting human faces in images or image classification. In this thesis, we will look into one problem of interest of computer vision, namely *facial feature detection*, from a specific viewpoint that has been hitherto mostly ignored for the most part: to efficiently and accurately detect facial feature points, such as nose, mouth or other features on users' smartphones and designing and implementing a pipeline of algorithms that is capable of achieving this in real-time by making full use of the phone's GPUs and thus parallelism.

In the following Chapters, the motivation, background and importance of this problem will be provided, along with a formulated research goal and the contributions of this thesis.

1.1 Background

Facial feature point detection is a very active area in computer vision [4] [6] [7] due to its many applications in consumer applications. Popular examples include accurately placing masks over people's faces e.g. in augmented-reality (AR) scenarios [8], visual speech recognition and emotion recognition [9] or real-time face-to-face transfer of facial expressions from one person to another [10]. Even though recent state-of-the-art methods have produced impressive results with regards to accuracy, there is a lack of real-time methods that estimate facial feature points [7]. Furthermore, there seems to be a great divide between devices commonly used in research (mostly desktop computers and desktop-grade GPUs) and devices where those algorithms find the most consumers in recent years, i.e. smart phones [11]. Given recent advances that enable these compute intensive tasks to run locally on the phone, combined with privacy-related issues when computing power is shifted towards the cloud, smart phones are taking up a central role in the future for the research area of computer vision, particularly regarding performance and battery efficiency. In contrast to research, industry giants like Google, Snapchat or Apple are working to continuously provide and integrate these algorithms in their operating systems and software development its (SDK), but at a price: deep lock-in into proprietary software with almost no open-source or research contributions, particularly with regards to facial feature detection.

This circumstance provides the underlying motivation for this thesis: to study computer vision algorithms central for efficient real-time computing of facial features from a smart phone perspective. The goal is to provide a *real-time* implementation achieving state-of-the-art performance in facial feature detection. This short introduction into the domain will be followed by the formulation of the research question of this thesis as described in the Section 1.2 .

1.2 Research Goal

The research goal of this thesis is to methodologically design, study and implement a **real-time facial feature detection pipeline** running on the **iOS platform** while achieving state-of-the-art accuracy as compared to research in the area. In particular, the goal is the evaluate or design algorithms that contribute to the performance aspect of this pipeline and can be exploited and run **entirely on GPU** of the device, and thus provide a reference implementation outperforming available solutions in terms of runtime performance.

1.3 Scope and Setup

For the implementation of the pipeline, a raw RGB image stream from a smart phones camera will be considered as only input to the subsequent stages of processing. Only the scenario of accurately detecting the features of a *single* face will be examined, thus certain aspects will implemented not generically and thus to only work under this circumstance.

Given the vast amount of available literature in the field, only a small subset of potential algorithms will be reviewed throughout this thesis in terms of consideration for the pipeline implementation which have been pre-selected mainly for the criteria of parallelism and implementation difficulty.

We emphasize that all chosen algorithms were implemented by the author of this thesis to run on the phone, and no external libraries were used in this process - mostly because there are almost none existent specifically designed to be run on the phone (and on the phone's GPU in particular) - except for open source datasets and pre-learned classifier and regression parameters.

The hardware setup for development and experimentation is a standard consumer-grade iPhone 7 using the front-facing camera with $720p$ resolution as input unless otherwise specified. Desktop computers were only used for offline-training of (machine-learning) algorithms.

The pipeline is designed to be run on the iOS platform, which was chosen due to its very-well documented Metal API, a C++ based shader language, along with a light-weight interface to execute commands and computations on the phone's GPU (see Section 2). Given lack of features for the required task in OpenGL[1] versions found on Android devices and the yet experimental stage of Vulkan[2] on mobile phones, the iOS platform seems the best choice at the moment for experimentation of GPU-based algorithms as well as machine learning.

Throughout this thesis a number of experimental benchmark results will be provided which were run repeatedly to give accurate average results in terms of frame- and runtime. Popular image datasets from literature were used to cover a range of scenarios that would apply to a production scenario when the pipeline would run on a phone. Experiments using video material as input - apart from the real time video processing using the phone's camera directly - were considered out of scope of this thesis.

1.4 Contributions

- **High-performance real-time facial feature detector pipeline** In this thesis the currently - to our knowledge - fastest available open source facial feature detection algorithm was developed that is available on the market for mobile phones. Building on a number of custom approaches to efficiently bring haar-like cascade face detection on the GPU, the designed pipeline makes use of parallel regression forest computation to speed up facial feature detection on the phone, paired with a custom Lukas-Kanade optical flow implementation to further improve detection speed. The final setup achieves an average of 25.0506 ms per frame or 39.9 FPS using a $720p$ input resolution on an iPhone 7. It further achieves an average error of 0.1977 of facial feature points normalized by interocular distance over a large popular facial feature dataset, making it comparable to state-of-the-art results on the used dataset [4].

- **Performance Study** An extensive performance study is included in this thesis that highlights the performance implications of different aspects, modules and stages of the pipeline along with explanations and areas of further improvement for future research.

- **Redesign and parallel implementation of computer vision algorithms for the iOS platform running on GPUs** Throughout this thesis, a number of key computer vision algorithms are dissected and redesigned to be efficiently run on the GPU in a mobile setting. A number of sub-problems in different stages, such as efficient rectangle grouping, were rebuilt with best practices in mind and are to our best knowledge new approaches to how computations can be solved in a single GPU commandbuffer without any dependency on the device's CPU for raw computation.

[1]`https://www.opengl.org/`

[2]`https://www.khronos.org/vulkan/`

1.5 Organization

The remainder of this thesis is organized as follows:

- Chapter 2 gives an introduction into the basics of making computations on the GPU in general, along with the basic principles of GPU architecture with focus in the iOS device and shader language.
- Chapter 3 describes a number of key algorithms that were selected and evaluated for the implementation of the facial feature detection pipeline, along with a reasoning on why these algorithms were selected in particular and how their realization would benefit a GPU-only based approach.
- Chapter 4 gives a detailed insight into the implementation of the pipeline, explaining all underlying principles involved in the chosen algorithms and how individual submodules were maximized in terms of achieving real-time performance.
- Chapter 5 evaluates the designed pipeline by looking into detailed performance and accuracy studies using publicly available datasets.
- Chapter 6 concludes the thesis mentioning notable results and giving an outlook into potential points of improvements and future areas of research.

2 iOS Platform and GPU Basics

2.1 GPU-Computing

Driven by the market demand for realtime high-resolution 3D graphics processing on the one hand [12], and the increased need for general purpose processing on the GPU driven partly by machine learning on the other, Graphic Processor Units or GPUs have evolved into highly parallel processing tools containing high memory bandwidth on both desktop and mobile. Compared to the CPU, GPUs are designed such that many more transistors are devoted to data processing thus making it possible to achieve enormous speedups in both industrial and scientific areas [13] wherever tasks can exploit high levels of parallelism. Whereas GPU cores historically inhabited fixed-functionality for a specific purpose, modern GPUs moved towards a fully programmable pipeline where individual cores can be programmed and addressed individually. In the following Sections we will depict the main concepts used in GPU-based programming which to a large extent apply to all types of GPU-architectures, frameworks and programming languages available on the market today. As we make extensive use of general-purpose based utilities such as compute shaders throughout this thesis, we will limit our description to these concepts and consider it out of scope of this work to detail typical 3D-GPU pipelines using e.g. fragment or vertex shaders.

2.1.1 Host & Device

One of the key concepts, when designing pipelines to be run on the GPU, is the computational and memory-based differentiation between CPU and GPU. The CPU, generally referred to as Host, and the GPU referred to as Device in the following, exhibit separate memories, where each processing unit may efficiently access only its own memory [14]. Thus, data required by one of the processing units residing in memory of the other must explicitly copy data between the two memory spaces, a process generally referred to as *Managing Communication* [14]. As a rule-of-thumb, this copying step is considered expensive in terms of execution time due to bandwidth and latency limitations [15], thus efficient algorithms try to minimize regular interactions between host and device and to generally limit e.g. the number of bytes transferred to a minimum in order to save e.g. frame time and to save time required for memory synchronization between the two devices.

2.1.2 Kernel, Threads, Blocks & Grids

In essence, the GPU can be seen as a device capable of executing a very large number of threads in parallel [16]. Most GPU languages allow developers to write kernels in specific shader languages, i.e. small programs that run on the GPU and execute a set of instructions. At the highest level, each kernel creates a single grid consisting of many thread blocks usually arranged in two dimensions [15]. Due to the SIMD (single instruction multiple data) paradigm of how threads are processing on the GPU, each thread block is assigned a SM (Streaming Multiprocessor) for its execution, containing a specific number of threads which can be to some extent controlled by the developer. Threads within a thread group share specific resources, such as shared memory, and are launched or scheduled in groups themselves, typically referred to as warps. Thus, a single thread on the GPU can be categorized using a three-level-hierarchy [17]:

- Grid - a set of thread blocks executing a kernel function
- Thread Block - a set of hundreds of threads that can share certain memory
- Warp - a batch of threads executed by the SM at the same time (typically 32 threads)

In Figure 2.1 [1], an example is shown how an image could be processed in terms of grid and blocks. Assuming each invoked thread for our kernel is processing a single pixel of the image or texture individually, the programmer needs to determine the kernel should be launched in terms of grid and block size.

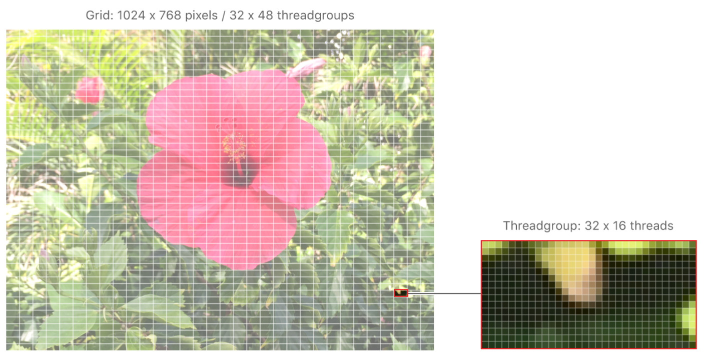

Grid: 1024 x 768 pixels / 32 x 48 threadgroups
Threadgroup: 32 x 16 threads

Figure 2.1: Grid Size and Threadgroups

In the example, our image has a resolution of $1024 \times 768 = 768432$ pixels. Given the goal of maximum utilization/minimum underutilization, we use a threadgroup size of $32 \times 16 = 512$ threads and a total number of $32 \times 48 = 1536$ threadgroups to schedule a total of $1536 \times 512 = 768432$ threads, thus having exactly one thread processing one pixel. While there are generally no limitations with regards to maximum number of threadgroups scheduled (except potentially processing time limitations on the GPU), the maximum number of threads per threadgroup is hardware dependent. While on an iPhone 7 the maximum threadgroup size is 512, the newer iPhone X uses a maximum threadgroup size of 1024 for instance, thus making it possible to process twice the number of pixels while scheduling the same amount of threadgroups.

[1]See Calculating Threadgroup and Grid Sizes

Most architectures only allow (or are optimized to) scheduling a specific number of threads depending on warp size, leading to underutilization.

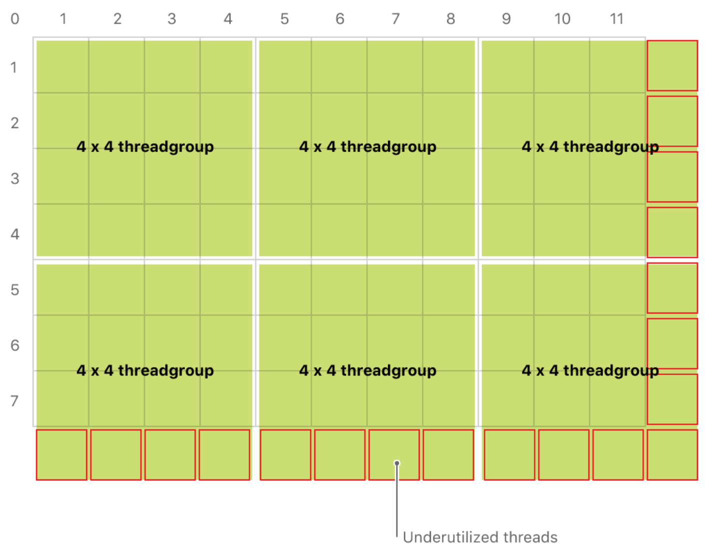


Figure 2.2: Grid Execution and Underutilization

In Figure 2.1 [2], an example is shown where to total number of threads scheduled exceeds to the number of required threads for computation due to an uneven grid size. The developer needs to optimize kernel launches for this circumstance and needs to check for out-of-bound memory reads or writes in the respective kernel. In the following we describe the two most important memory types/abstractions used throughout this thesis.

2.1.3 Memory Types

Global Memory

Typical GPU architectures expose a general-purpose, random-access, uncached off-chip memory [15] typically referred to as global memory. This memory type is visible to all threads launched on the GPU in a kernel for instance, and it is in many ways similar to a CPU's memory in generability and size [15].

Shared Memory

Shared memory (or threadgroup memory) on the other hand is a very small and fast on-chip memory, which is typically uninitialized [17] but can be accessed very rapidly (comparable to register access) by all threads in a single threadgroup. Each launched threadgroup has its own shared, independent memory. This is very useful and heavily utilized whenever threads need to share information, e.g. in convolutional shaders where often the sum of a pixel area is required. Most APIs provide synchronization utilities between threads, e.g. to make sure all threads have written or read into shared memory by providing synchronization points.

[2]See About Threads and Threadgroups

2.2 iOS Metal Shader Language

2.2.1 Overview

In the following Sections, we will briefly discuss the Metal Unified Graphics and Compute Language referred throughout this document as Metal. Metal is a C++-based programming language that iOS and MacOSX developers use to execute graphics and data-parallel computations running on GPUs [18]. Metal works together with the Metal Framework, which is the counterpart of Metal on the Objective-C/Swift Side managing allocation of GPU-based memory, execution of shader pipelines and e.g. display of visual information on the respective display. Metal is integrated into the normal iOS/MacOSX build pipeline using clang and LLVM rendering it easy to use shaders in common iOS applications. Metal supports a subset of the C++14 specification, thus projects have emerged[3] to provide i.e. cross-compiling capabilities against other emerging coding languages for GPUs such as Vulkan or CUDA [12]. Metal was specifically designed to have a low operation overhead [19] or - in other terms - a very thin API as compared to OpenGL for example. This makes it possible to decrease both CPU and GPU time for shader execution [20] and has consistently shown better performance in experiments [19].

2.2.2 Metal Objects

Figure 2.3 gives an overview of key objects utilized within metal.

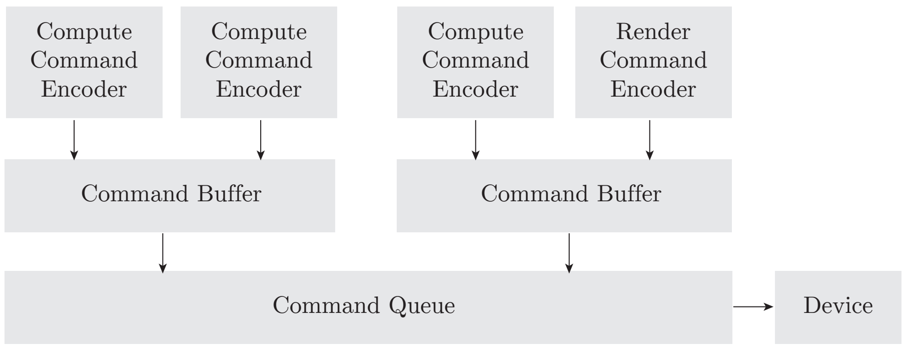


Figure 2.3: Metal Command Submission Overview

The *Device* is an abstract representation of a single GPU existing throughout the lifetime of a program. The *Command Queue* is an abstract object that can queue a serial (ordered) sequence of command buffers that are to be scheduled on the GPU. Command Queues are thread-safe and allow multiple command buffers to be encoded simultaneously [4]. A *Command Buffer* contains a set of *Command Encoders* specifying kernels to be executed on the GPU and thus store a set of hardware commands. *Command Buffers* need to be commited for execution, and in case results of those computations are required on the CPU-side, developers need to either synchronously wait for its completion or asynchronously check the completion status, which is typically realized using callbacks. Non-blocking implementations typically

[3]See for instance MoltenGL https://moltengl.com/
[4]See for more information about Command Queues

have the advantage that subsequent command buffers can be already encoded on the CPU while the GPU is still processing - thus achieving higher GPU utilization. *Command Encoders* convert developer-specified API commands into hardware commands using three different types of command encoders: render encoders, compute encoders and blit encoders. Note that different types of command encoders can be interleaved into a single command buffer [19]. Render encoders are special types of kernels used for rendering graphics content onto the user's display, while blit encoders are used to copy buffer or texture data. Compute encoders, which we make extensive use of in our own pipeline, are general purpose kernels that can be used to process data.

2.2.3 Shaders & Argument Tables, Example

In the following we will give a brief example of a shader and its argument table, as well as the opposite host code to call this kernel. As stated before, Metal shader language is a subset of C++, thus the syntax in Code 2.1 should look familiar to C++ developers.

Code 2.1 MSL Example Shader

```
kernel void example_kernel(
    device packed_float2* A [[ buffer(0) ]],
    constant float &B [[ buffer(1) ]],
    texture2d<float, access::write> C [[ texture(0)]],
    texture2d<float, access::read> D [[ texture(1)]],
    uint2 globalIdx [[thread_position_in_grid]]
) {
}
```

A kernel is a single function that consists of a number of parameters called the Argument Table [19]. Each of the inputs and outputs to the kernel needs to be specified in the argument table, and access qualifiers determine how, if and in which frequency data can and should be read or written to in the kernel. In the example above, we have four entries in the kernel's argument table. The first two entries are references to buffers: in the first case A, we give the kernel a pointer to buffer residing on global memory which it can read from and write to using a floating point datatype. Metal data types have different byte alignments in general, using the keyword *packed* we can ensure that individual floating point values reside in subsequent memory. In this case, developers need to ensure themselves that no reads or writes are made that are out-of-bound of the buffer. In the second case B, we provide a single constant floating point value, which is under the hood copied to constant memory of the kernel. In cases C and D, we provide textures that require access qualifiers, specifying whether the texture will be read from or written to. It is possible to provide the same texture twice to the kernel, once for reading and once for writing, given that at least on the iOS platform there is no qualifier for simultaneous reading and writing. Apart from I/O entries in the argument table, we can supply kernel function input attributes that represent utility variables, e.g. to find the current threads unique position in the grid upon which it is executed [18].

Code 2.2 Example Shader Call

```
let commandBuffer = queue.makeCommandBuffer()
let encoder = commandBuffer.makeComputeCommandEncoder()
encoder.setBuffer(buffer, offset: 0, index: 0)
encoder.setBytes(&A, length: MemoryLayout<Float32>.stride, index
   : 1)
encoder.setTexture(C, index: 0)
encoder.setTexture(D, index: 1)
encoder.setComputePipelineState(pipeline)
encoder.dispatchThreads(threadsPerGrid, threadsPerThreadgroup:
   threadsPerThreadgroup)
commandBuffer.commit()
```

In Code 2.2 , we demonstrate the use of the Metal Framework using Swift in order to execute the kernel described previously. First, we instantiate a *MTLCommandBuffer* object which will instruct the GPU to process our kernel. Next, we create a *MTLComputeCommandEncoder* object from our command buffer, and set the required arguments exactly as they are specified in the kernel's argument table. Then, we use a *MTLComputePipelineState* object, which represents our already compiled shader function to specify the kernel function to be launched in the compute encoder, and dispatch the number of threadgroups and threads per threadgroup we require. Finally, notify the API that we have completed encoding commands to be command buffer and that it is ready for execution on the GPU. Throughout this thesis, we will describe a multitude of different shaders using this structure which only comprises a small overview of possibilities the Metal API has to offer. For further reading and understanding, we refer the reader to the excellent Metal Shading Language Specification [18].

3 Pipeline Overview & Algorithmic Selection

3.1 Overview

The goal of this thesis is to describe, implement and evaluate a high performant facial feature detection library using the iOS platform. One of the key subgoals of this thesis was to design a pipeline that is residing entirely on the GPU, thus eliminating expensive communication between host and device.

In this Section, we will describe a number of key algorithms in the field of facial feature detection and computer vision in general that were selected in our pipeline as well as the underlying reasoning for choosing those technologies in particular, especially with regards to the potential level of parallelism that can be exploited.

3.2 Facial Feature Detection Algorithms

Facial feature or landmark detection in general is a well-studied problem in computer vision [7] and aims to localize a set of semantic points, such as nose tip, eye-corners, etc. of a face within an image in an efficient and accurate way [6] [4].

It is widely used for a multitude of problems, such as facial express analysis, head-pose estimation or gaze tracking and a variety of different techniques exist to approach this problem. Roughly, classical literature on the topic can be divided into two main areas: holistic methods and methods that use local features [6]. Holistic methods use the texture over the whole face region in order to fit a generative model to a test image [7], while local-feature based models often use regression techniques to iteratively adjust facial landmarks in a cascaded way starting from an initial estimate [21]. The following Sections give a brief overview of key techniques used in the field.

3.2.1 Active Appearance Model

Active Appearance Models (AAM) as proposed in [22] are among the most commonly used and most popular facial feature detectors. Originating from the idea of Active Shape Models [23], they are non-linear, generative and parametric models that describe shape and appearance of a face separately [1].

The shape of a face is described by a mesh of v vertices allowing linear shape variation.

In Figure 3.1 , the base shape of a face is described by s_0, which is usually the average face of a training set of images. By allowing linear combinations of individual shape vectors $s_1 - s_3$ describing linear deformations of the face, we can capture a large variety of different

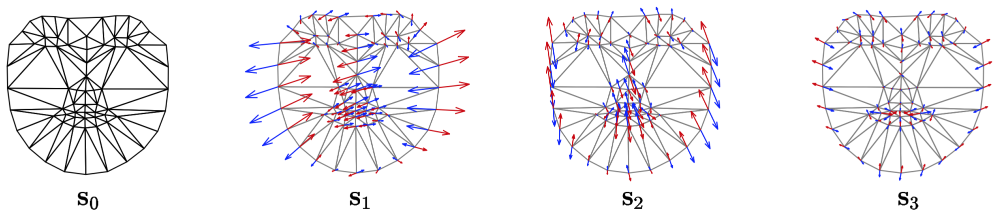


Figure 3.1: Linear Shape model of an AAM, from [1]

shape types of faces. This, the shape s of a face can be expressed as a base shape s_0 plus a linear combination of n shape vectors [1]:

$$s = s_0 + \sum_{i=1}^{n} p_i s_i \tag{3.1}$$

Thus, having computed s_0 and s_i during training time, one can estimate the coefficients p_i online to gather the final shape of a face from an image. In practice, the vectors s_n are the eigenvectors corresponding to the n largest eigenvalues [22] gathered from a training set of images and are computed using principal component analysis (PCA) after normalization. Similar to the face's shape, the appearance of an AAM can be defined as a base appearance in form of an image A_0 plus a linear combination of m appearance images A_i [1]:

$$A = A_0 + \sum_{i=1}^{m} \lambda_i A_i \tag{3.2}$$

After normalizing the training images, the appearance images A_i can be again computed using PCA. Computing λ_i and p_i can then be formulated as a non-convex optimization problem, where typically a texture sample is used to compute an error vector in order to iteratively refine predicted displacements. Given our goal of providing a GPU-based implementation, holistic approaches such as AAM or ASM are difficult to parallelize and computationally intensive, rendering them unsuitable for the purpose of this thesis.

3.2.2 Regression-based Methods

Regression-based methods have recently gained momentum in academic literature due to their high fitting speed and accuracies they achieve under the right conditions [24]. In general, an initial face shape s_0 determined from a training set is iteratively aligned to a face image by applying a shape increment $\vartriangle S$. $\vartriangle S$ on the other hand is determined using (a cascade of) regression functions [24] on the input image, typically on local texture features around each landmark. Notable examples of this technique include Valstar et al. [25] which use support vector regression or Cao et al. which [26] employ cascaded fern regression. Another very popular approach are random regression forests based on local image patches as in the famous paper by Kazemi et al. [21]. These approaches are interesting in the context of this thesis as they exhibit parallelism by nature: individual forests or trees (e.g. belonging to different features) can be computed separately, thus making them good candidates for our pipeline in terms of potential performance and overall lower complexity. We will thus focus on the work of Kazemi et al. and further investigate their approach of using an ensemble of regression trees and how this could be realized on a mobile GPU.

In the approach by [21], a cascade of t regressors r_t is trained on a set of images to iteratively update a base shape S_0, where each regressor is a function of the previous shape S_t and the grayscaled image I. In each cascade, a random forest consisting of a large number of decision trees is evaluated independently, where each leaf node in the tree contains relative deltas in x and y direction of the respective landmark in relative shape space. Decisions on how a tree should be traversed are made by comparing relative pixel intensities. These two circumstances introduce some form of geometric invariance on the one hand [21], but also a certain invariance to change in lighting conditions due to only relative pixel comparisons in contrast to absolute thresholding values. Additionally, only a sparse representation of the image is required, as only specific pixel positions are required for the decision trees, making it very computationally efficient to compute and ideal for a GPU-based implementation, where on the one hand pixel values can be gathered individually by threads, and decision trees can be individually computed by threadgroups. Thus, this approach was selected for implementation and is further detailed in Section 4.4 .

The main disadvantage for this approach stems from the fact that it requires a good initialization of the base shape, e.g. using an accurate bounding box of the face to initialize the shape within the face region. In the following Sections we will detail two popular approaches to this problem.

3.3 Object & Feature Detection

In order to be able to obtain a good baseline initialization for average facial feature shapes, as described in Section 3.2.2 , an accurate detection of the face's location is required in the image. This task is generally known as object detection and is a well-researched area in computer vision. The main task of an object detector is to retrieve a tightly-fitting bounding-box around object(s) of interest in an image, often along with a probability measure or classification score when for instance using detectors for different types of targets. The key challenge is the high amount of variation in the occurring in the form of position, rotation, occlusion, distance, lightning, etc., which good object detectors need to be able to deal with.

Generally, the task of object detection can be split into two different parts [27]:

- Feature extraction to compute a number of image descriptors or feature vectors
- Detection frameworks that make use of these descriptors to detect objects

For the purpose of this thesis, a subset of of the available literate and methodologies will be reviewed that are most suitable to be applied to a scenario running on a mobile GPU - in particular we will emphasize *local* image descriptors, which focus on different parts of the image rather than viewing the image as a whole as in the case of *global* image descriptors due to better computational efficiency in general. In addition, only those extraction techniques and detection frameworks will be highlighted which are most prominently used in the area of face detection.

3.3.1 Feature extraction

Haar-like features

Originally introduced by Viola et. al [28] and then futher extended by the research in [29], haar-like features are differences of rectangular regions arranged in different scales and positions within in image [27]. Figure 3.2 shows five examples of different haar-like patterns.

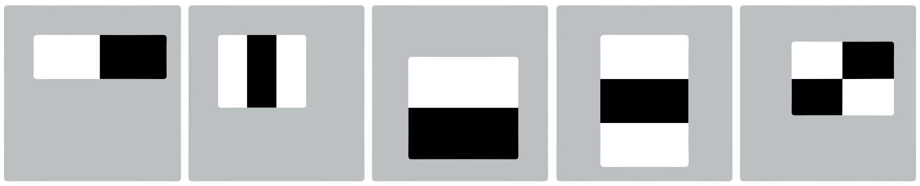


Figure 3.2: Five haar-like patterns as in [2]

Given a mean and variance normalized image, the idea is that different templates of shapes, sizes and forms of haar-like features can give indications about the characteristics of the underlying object in the image. In Figure 3.2 , the sum of pixel values indicated by the white area would be deducted by the sum of pixels in the black area to compute a single haar-like feature. These features can be applied at different images scales additionally to give scale invariance or to capture different details of the image. It should be noted that the authors of [28] also introduced the idea of an integral image, i.e. a precomputed cumulatively sum-table that makes it computationally very efficient to compute the sum of pixels in an area, making this approach very interesting for applications on the GPU, as each haar-like feature can be computed individually and efficiently.

Local binary patterns

Local binary patterns (LBP) are a popular method to extract useful features from images and are widely used in different areas. First proposed by Ojala et al. [30], they are a ordered set of binary comparisons of pixel intensities between a pixel center and its surrounding pixels [31]. Figure 3.3 gives an example where the black dot indicates the center pixel, and the gray pixels indicate the surrounding reference pixels.

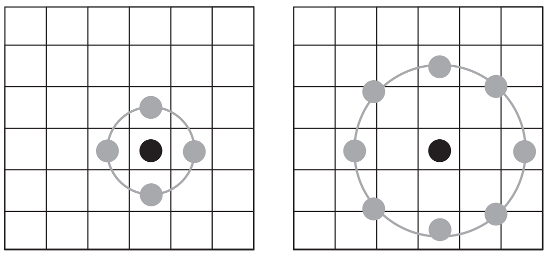


Figure 3.3: Local Binary patterns

LBP was originally defined by 3×3 neighborhoods and thus eight surrounding pixels, and was then encoded using 8 bits where each bit would indicate whether the pixel intensity in the center was greater than or equal to the the comparison pixel. Different forms, shapes or ring sizes have been suggested for different problem settings since the authors proposed the LBP operator. This form of feature extraction is also well suited to be performed on the GPU, as individual threads of the threadgroup can process pixel comparisons in parallel.

Histograms of oriented Gradients

Histograms of oriented gradients (HOG) as introduced by [32] represent another popular technique for feature extraction. The basic idea behind HOG is that distributions of local intensity gradients in grids or cells within an image can be well-used to characterize local object appearance. In order to compute HOG descriptor blocks, an image is thus first divided into small spatial regions or cells. Within each cell, a local 1-D histogram of gradient directions or edge directions over the pixels of cells is computed [32] after usually having performed a contrast-normalization step over larger image areas in order to get better invariance to illumination etc. Given that the normalization step is usually expensive over larger areas of the image, this approach is less suitable for selection in the pipeline.

3.3.2 Detection and Classification Methods

Support Vector Machine Classifiers

Support vector machine (SVM) classifiers have been very successfully applied to the task of object detection in computer vision [27] in previous research in combination with HOG descriptors for instance. Formally, SVM build a predictive model using supervised learning by finding a hyperplane that maximizes a margin of two different point sets, which can then be used for classification, e.g. detecting whether the image contains an object or not given a number of n-dimensional feature descriptors. Different approaches have been made using SVM to detect faces, ranging from simpler approaches such using a tiled detection window to combine HOG features into a larger feature vector in a SVM window classifier as in [32], to hierachically clustering different separate individual object part descriptors and then learning SVMs for each cluster [33]. SVM performance at test-time highly depends on the individual setting and typically suffers from scalability problems as it is hard to parallelize, even though research has been made on the topic in a distributed setting [34].

Cascaded Approaches and AdaBoost

Introduced in the context of haar-like features [28], the boosting machine learning paradigm proposes to determine the most informative haar-feature templates for a specific classification task. In particular, these approaches usually learn a cascade of collections of classifiers, where each stage in the cascade can be used to reject improbable areas in the image where e.g. no face is detected. Different alterations have been tried using this approach, e.g. directly estimating facial features from haar like features [35], or more elaborate classification models using multiple classifiers on the statistics of localized parts better capturing geometric relationships between object parts [36].

Although these types of cascades are rather slow to train, they offer significant improvement as compared to SVMs in the run-time of the final detectors [27], making them more ideal to use in our GPU-based pipeline. The general coarse-to-fine strategy of having weak classifiers to early discard invalid areas of the image seems well-suited for a fast detection pipeline, particularly when combined with haar-like features which can be evaluated very rapidly. A more detailed description of the details of the implementation of the evaluation of cascaded haar-like features will be provided in Section 4.3 .

3.4 Optical-Flow & Tracking

In the previous Chapters, a particular emphasis with regards to highlighted algorithms was placed on stationary analysis of frames. However, given that a continuous video stream is typically analyzed when facial features are determined over time, it makes sense to make use of algorithms that are capable of using the temporal component in the desired pipeline. Rather than trying to detect e.g. the face or facial features in each frame, one could analyze the first frame of the video stream and then try to estimate their position(s) in the subsequent frames by employing a tracking mechanism, followed by constant re-detection of the face in case of drift.

One of the most popular and well-researched methodology to achieve the above-mentioned problem-setting is optical flow. Given a series of time-ordered images, the goal of optical flow is to determine the pattern of 2D-motion in terms of displacement of features from one image to another, where the displacement per pixel or selected feature is given by x and y offsets with respect to time [37].

In general, two different types of optical flow can be differentiated:

- *Sparse* optical flow - determine optical flow only for specific pixels selected in the image
- *Dense* optical flow - determine optical flow for all pixels in the image

Both types have different applications and solutions, for the desired pipeline however, a stronger emphasis will be placed on sparse optical flow as we only required tracking of facial feature points or the face bounding box rather than the entire image. In the remainder of this Chapter, two of the most popular approaches to optical flow are explored.

3.4.1 Lucas-Kanade Method

The Lucas-Kanade appproach, as originally specified in [38] is counted among the sparse optical flow methods. In particular, it assumes that images can be described by a displacement vector for each pixel of the image from one frame to the next, that this (small) displacement vector is a function of the position x and that the optical flow is constant in the local neighborhood for a specific pixel of interest [39]. This is especially valid when looking at very small intervals between images, e.g. high frame rates of the video.

By using the optical flow equation based on partial derivates with respect to spatial and temporal coordinates and assuming the above mentioned assumptions, one can solve using the least-squares principle and obtain the displacement vector within a local neighborhood or window of interest. The reader is referred to a more detailed description of this algorithm which will be provided in Section 4.5 .

Several improvements have been suggested to the basis of this idea interesting to our application. By applying the Lukas-Kanade solution to an image pyramid for instance other than an image at a single scale, one can capture coarse motions in addition to fine motions only visible at larger scale (i.e. downsampled images) [38]. In addition, the least squares method can be performed iteratively on each pixel multiple times in order to accumulate flow in each iteration until a certain magnitude threshold is reached, making it a suitable candidate for fine-grain parallelism and use on the GPU [40].

3.4.2 Horn-Schunck Method

Another popular method introduced by Horn et al. [41] argues that optical flow cannot be computed locally and assumes smoothness in the flow over the whole image, introducing a smoothness constraint globally.

In contrast to the Lukas-Kanade method, a global flow function is introduced and minimized in order to account for distortions. Equations central to this problem can be solved using iterative procedures and but in general require more computational effort than local methods such as the Lucas-Kanade method. In addition, literature has found that the Lucas-Kanade method reduces noise better as compared to the Horn-Schunck method thus giving more accurate and more importantly for our thesis, computationally performant, results [42]. This method is thus disregarded due to performance implications.

3.5 Implemented Pipeline

After having described the key algorithms from literate which can be utilized to build a potential pipeline for facial feature detection on the phone, this Section will describe which components were selected and how they work together. As established in Section 3.2.2 , a regression based method for facial feature detection seems most natural and performant to be implemented on the phone's GPU. For this purpose, the cascaded approach of [21] will be as one of the key ingredients of the pipeline using random regression forests based on local image patches. A custom algorithm implementing this idea will be built aiming at maximum utilization of GPU threads in the pipeline. In order to retrieve a good initialization of base points for this regression-based implementation, we chose a combination of extraction of haar-like features in combination with another cascaded classification approach using a tree-like classifier with weak and strong classifiers as in [28]. By providing a performant integral-image computation implementation, haar-like features can be very rapidly evaluated individually by threadgroups or threads and thus make this approach very suitable for our overall *real-time* target. This undertaking will required a number of algorithms that will be re-evaluated in terms of usability on the GPU, which will form part of the contribution of this thesis. Finally, a Lukas-Kanade optical flow module will track found facial feature points and will deform the found facial bounding box of previous frames using a similarity transform. The overall process in shown in Figure 3.4 .

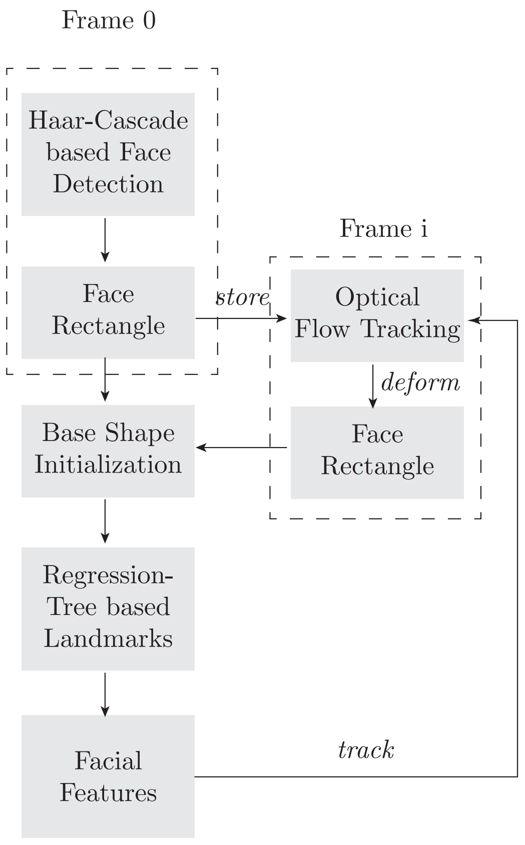


Figure 3.4: Overview of chosen pipeline

The overall goal of this pipeline is to be residing entirely on the GPU in order to avoid expensive host communication, and to provide a single-commandbuffer implementation that can be run in a performant-way achieving our real-time target.

In the next Chapter 4 , more light will be shed on the functionality of this chosen pipeline, as well as how individual components were approached, implemented and then further refined.

4 Implementation

4.1 Overview

This chapter gives a detailed overview into the implementation details of the pipeline described in Section 3 . As mentioned before, one of the key goals during implementation was to achieve a pipeline entirely residing on the GPU in order to avoid expensive transfer of data between host and device on the one hand, and to be able to send a single set of instruction commands without the need to wait for completion of intermediate results. As described in Section 2 , this was achieved, among other factors, using a GPU-based buffer as dispatch argument for certain kernels, i.e. handling further processing logic directly within compute shaders. Thus, results, such as whether a face was detected in a video frame, can be easily passed on to subsequent modules on the pipeline by setting the respective compute shaders' block and grid dimensions accordingly. The remainder of this Section goes into further detail of each pipeline module. We start in Section 4.2 by describing a set of basic algorithms as well as their implementations on the GPU, which will be used in multiple places throughout the full facial feature detection pipeline, followed by the integral image computation as required by the haar-cascade based face detection in Sections 4.2.3 and 4.3 . We then depict the facial feature detection algorithm and its parallel implementation in Section 4.4 , and finish by showing our implementation of the iterative Lucas-Kanade optical flow algorithm [38] in Section 4.5 .

4.2 Basic Algorithms and their GPU implementations

4.2.1 Parallel Prefix Sum Scan

One of the most common building blocks of a large number of GPU-based algorithms is the so-called prefix sum operation [43]. The original definition as per Belloch et al. [44] is as follows: The operation takes a binary associative operator $\oplus$, and an ordered set of n elements

$$[a_0, a_1, \ldots, a_{n-1}] \tag{4.1}$$

and returns the ordered set

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})] \tag{4.2}$$

In general, the binary associative operator $\oplus$ can be defined as required, however, we will only describe the case where $\oplus$ is an addition, i.e. taking

$$[0,1,2,3,4,5,6,7] \tag{4.3}$$

and returning

$$[0,1,3,6,10,15,21,28] \tag{4.4}$$

as an example. The all-prefix-sums operation on an array of data is usually referred to as scan [43]. The above definition describes an *inclusive scan*, because elements up to $n = j$ are summed including the element a_j as compared to the case where a_0 for example would be either identity or zero (*exlusive scan*). Both *inclusive scan* and *exlusive scan* can be converted by shifting the resulting array either left or right and inserting the last sum or identity at the respective place [44]. The scan operation will be used at multiple places within the implementation, most prominently in stream compaction in Section 4.2.2 and the computation of the integral image in Section 4.2.3 .

A naive sequential scan computation might look as follows:

Algorithm 4.1 Naive Sequential Exclusive Scan Computation

```
1: procedure naiveSequentialScan
2:     $out[0] \leftarrow 0$
3:     for $i \leftarrow 1 : n$ do
4:         $out[i] \leftarrow out[i-1] + f(in[i-1])$
5:     end for
6: end procedure
```

This implementation however is not well suited to GPUs because it does not take advantage of data-parallelism [43]. Thus, in our implementation we follow the approach of Belloch et al. [44] using *balanced trees*. In general the problem is split up into two phases:

- Reduce-Phase - Traversing the tree from leaves to root to compute intermediate sums
- Down-Sweep Phase - Traversing the tree back from root to leaves to build the final scan values

In the *reduce or up-sweep phase*, a tree is layered over the data array, and the operator $\oplus$ is used to sum the vertices at the current layer. Following our example, we retreive the results as shown in Figure 4.1 .

In general, this *balanced tree* $n = 8$ leaves has $d = \log_2 n = 3$ levels, with each level d containing 2^d nodes. Each node then basically performs a single addition, thus the algorithm performs n additions on a single tree-traversal. This tree can then basically be mapped onto an array upon which our algorithm operates, as indicated by the variables $t_0 - t_7$. The following code example in MSL performs a general reduce phase where the block width is required to be at least $\frac{n}{2}$.

In this example, rather than writing directly to device memory we perform the reduce phase in threadgroup memory as computed values are intermediary only. The threadgroup

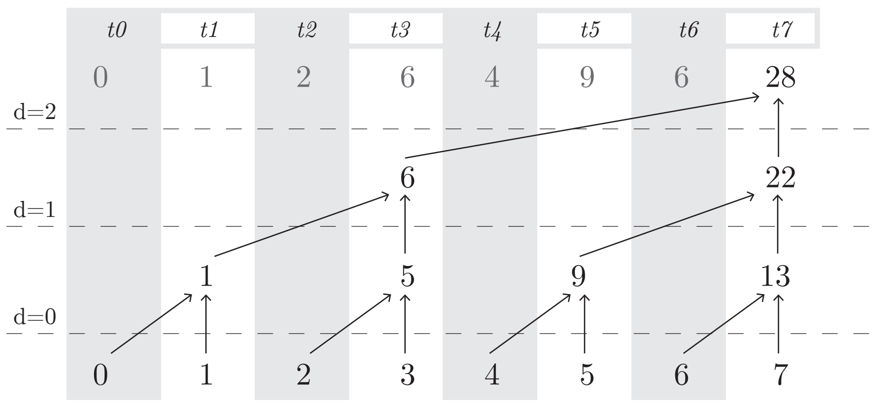


Figure 4.1: Reduce-Phase Balanced Tree

Code 4.1 MSL Reduce Phase

```
threadgroup float* temp[N];
temp[2 * threadIdx.x] = input[2 * threadIdx.x];
temp[2 * threadIdx.x+1] = input[2 * threadIdx.x+1];

int offset = 1;
int d, ai, bi;

for (int d = N>>1; d > 0; d >>= 1) {
        threadgroup_barrier( mem_flags::mem_threadgroup );
        if (threadIdx.x < d)  {
        ai = offset*(2*threadIdx.x+1)-1;
        bi = offset*(2*threadIdx.x+2)-1;
        temp[bi] += temp[ai];
    }
    offset *= 2;
}
```

memory can either be allocated directly within the shader using a constant N, or provided as an argument to the compute shader where the encoding instructions take care of allocating enough memory. In the first step, all threads are used to fill the values into the threadgroup memory - a single thread hereby performs two operations as only $\frac{n}{2}$ are issued in total. We then bitshift the number of elements to be scanned in order to retrieve the number of operations to be performed. In the example, we have $N = 8$, thus $d_0 = 8 >> 1 = 4, d_1 = 4 >> 1 = 2, d_2 = 2 >> 1 = 1$ operations. In each iteration, we synchronize all threads and indicate that we need to wait until all threads have finished writing to threadgroup memory, as subsequent operations further up in direction of the root of the tree require all earlier computations to be completed. Furthermore, we only utilize $t_N < d$ threads in each iteration, as the number of operations continuously decreases until only a single addition operation is left to perform in $d_2 = 1$. The variables a_i and b_i then contain the indices of the elements within the threadgroup memory to be reduced in the next stage.

In the *down-sweep* phase, we traverse the tree in the same fashion starting from the root.

We will describe the *exclusive* scan of this phase, i.e. the first element t_0 being 0 after the scan, and then show how to shift the array in order to retreive the *inclusive* version of the scan. Note that the last element in our example, t_7, contains the sum $\sum_{k=1}^{N} a_k$, i.e. the sum of all elements. This value will be needed later in order to shift the array. In the first step of the *down-sweep* phase, we zero the last element in the array as indicated in Figure 4.2 .

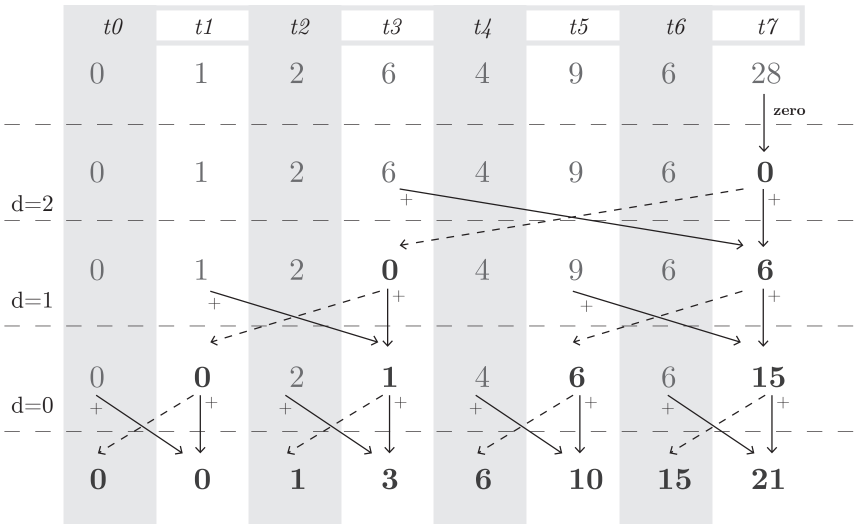


Figure 4.2: Down-Sweep-Phase Balanced Tree

After this step, we again iterate with the indices of d_N inverted in the iteration loop. Each iteration step performs two steps: first, compute the sum of two elements as indicated by the respective tree level d, and pass on the value of an element down the tree to the final leaves. In Figure 4.2 , this is shown by solid and dashed lines respectively. The further down we go the tree, the more of these operations are to be performed. After reaching leaf-level, the array contains the final result of the *exclusive* scan operation using addition. By shifting every element of this array one index to the left and inserting the previously mentioned sum t_7 at the end of the array, we now have the same result as in Equation (4.4) . The respective MSL code, shown in Listing 4.2 looks similar to Listing 4.1 , with the operations in the array and the iteration sequence modified accordingly.

First, a single thread (usually the first thread is utilized for similar tasks) blanks the last element of the threadgroup memory array. Next, in each iteration, we calculate the indices again - which are equivalent to the up-sweep phase - and perform our addition on the elements at the calculated indices. In addition, we replace the item at the lower index of the two with the value of item with larger index, as seen in Figure 4.2 . Finally, the values, now stored in threadgroup memory, are written back to device memory where they can be accessed by either other compute shaders or the CPU.

Large Array Scan

An obvious problem with this approach is the limitation of the maximum total threads per threadgroup, which is firstly depending on the complexity of compute shader functions and

Code 4.2 MSL Down-Sweep Phase

```
if (threadIdx.x == 0) { temp[N-1] = 0; }

for (int d = 1; d < N; d *= 2) {
        offset >>= 1;
        threadgroup_barrier( mem_flags::mem_threadgroup );
        if (threadIdx.x < d)  {
        ai = offset*(2*threadIdx.x+1)-1;
        bi = offset*(2*threadIdx.x+2)-1;
        float t = temp[ai];
        temp[ai] = temp[bi];
        temp[bi] += t;
    }
}

output[2 * threadIdx.x] = temp[2 * threadIdx.x];
output[2 * threadIdx.x+1] = temp[2 * threadIdx.x+1];
```

secondly currently limited to a maximum of 512 on modern iOS GPUs[1]. The solution to this problem is laid out in [44]: we simply divide the initial array into a number of blocks, according to the maximum threads per threadgroup, and write the individual sums to a separate, temporary array. Assuming a maximum number of threads of $t_{\max} = 2$, our example from this chapter would be processed as in Figure 4.3 .

The array containing $n = 8$ items is split into two individual blocks with $n = 4$ items. Each of these blocks is then scanned individually, and the respective sums are written out to an array of size 2. This can be scaled up to arrays and blocks containing more items - in case where even the individual array cannot be processed by a single block due to thread number limitations, one would need to perform this intermediary step again. The intermediary array containing the sums of individual blocks is then scanned, and the results are written back to the original scanned arrays. During this process, block n retrieves the results from the intermediary array at position $n - 1$, i.e. no sum needs to be added to the first block. The last step then simply merges the individual blocks back into an array of original size. Non power-of-two arrays can simply be handled by zero-padding the last block [43].

Subreduction & Warp Optimization

Given that threads in a threadgroup are issued in warps or SIMD branches (see Section 2), one can do additional optimizations in order to ensure that distinct SIMD processing elements execute different program paths after a branching instruction, which has shown to significantly improve performance [45]. Parallel reduction, or *subreduction* is a typical domain to this problem and is actually very similar to *scan* [46]. In our previous code, the if condition within the iteration of each loop leads to highly divergent warps [47]. Another problem is that upon certain iterations, a large number of threads are inactive as only a few threads make

[1]See https://developer.apple.com/metal/limits/ for a table of limits

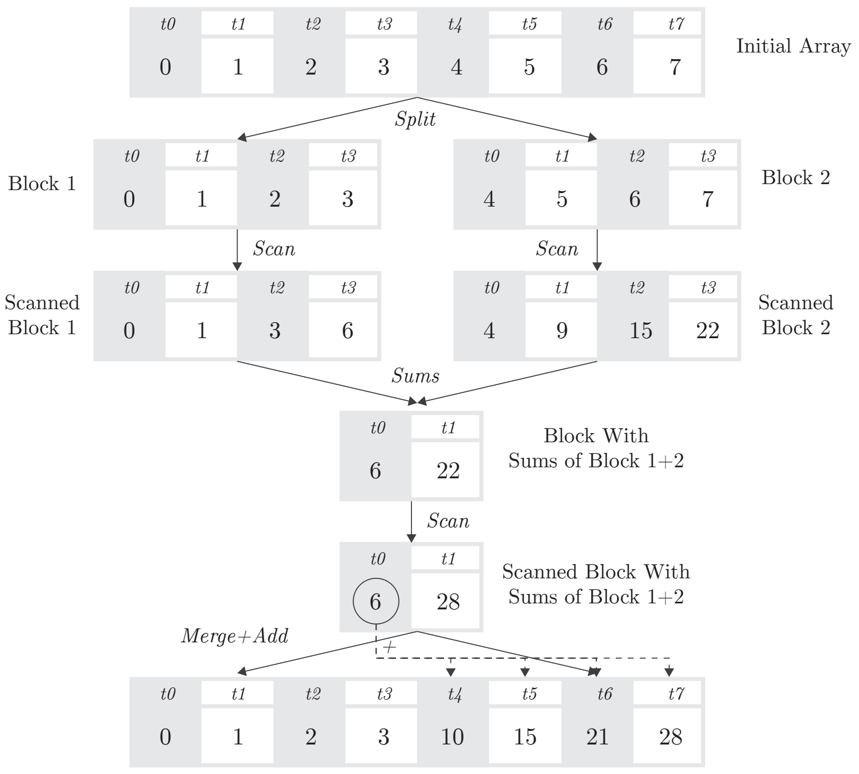


Figure 4.3: Scan of Large Arrays Example

computations. We will demonstrate an improvement on previous shader examples based on the *subreduction* problem, i.e. gathering the sum of a number of elements. First, given that the warp size is fixed for the iOS platform for the device upon which testing was performed during the implementation of this thesis (to NWarpSize = 32), we can unroll all loops and save threadgroup barriers within a single warp, as instructions are SIMD synchronous within a warp [47]. Listing 4.3 shows our production code used for an inclusive scan of values within a single warp.

We will later introduce the possibility of reducing or scanning multiple types of values, such as float4 containing four float values to further save memory bandwidth, thus for a number of functions we use a templated version. Given that MSL is a subset of C++, most functions can be templated in a similar way. The code in Listing 4.3 is unrolled for NWarpSize = 32. A more general version could easily be achieved using a loop once more. In general, this resembles the tree structure again with offsets being multiplied by two for each instruction, however, this time we save both the if condition and the threadgroup barrier command as we simply increase the threadgroup memory length to two-times the size of the number of threads.

Figure 4.4 illustrates the same function assuming a warpsize of 4, four instructions $I_1 - I_4$ and the values from Equation (4.3) . Thus, two warps are issued, each with 4 threads, both operating on a single shared threadgroup memory.

In the first instruction I_1, threadgroup memory is cleared to zero at positions `2 * threadIdx - (threadIdx & (4 - 1))`. Remember that threadgroup memory is uninitialized, thus this step is important to keep sums accurate and to not access random GPU

Code 4.3 MSL Warp Inclusive Scan

```
template<typename T>
device T warpScanInclusive(
        uint threadIdx,
        T iData,
        volatile threadgroup T *sData) {

    int pos = 2 * threadIdx - (threadIdx & (kWarpSize - 1));
    sData[pos] = 0;
    pos += kWarpSize;
    sData[pos] = iData;
    sData[pos] += sData[pos - 1];
    sData[pos] += sData[pos - 2];
    sData[pos] += sData[pos - 4];
    sData[pos] += sData[pos - 8];
    sData[pos] += sData[pos - 16];
    return sData[pos];

}
```

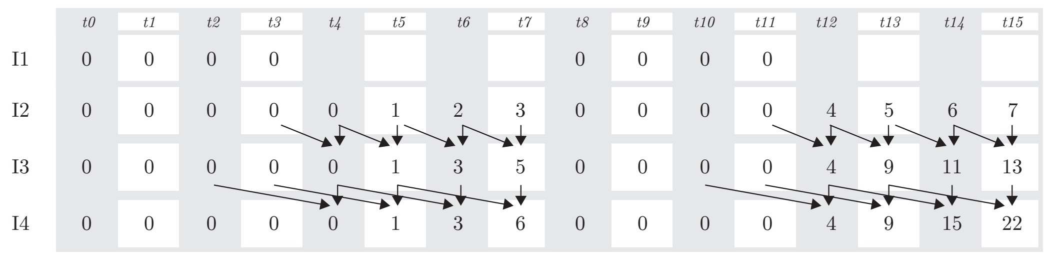

t0
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
t13
t14
t15
I1 0 0 0 0 0 0 0 0
I2 0 0 0 0 0 1 2 3 0 0 0 0 4 5 6 7
I3 0 0 0 0 0 1 3 5 0 0 0 0 4 9 11 13
I4 0 0 0 0 0 1 3 6 0 0 0 0 4 9 15 22

Figure 4.4: Warp Inclusive Scan

memory. Instruction I_2 then sets the actual values by shifting the positions by the value of the warp size. In instructions I_3 and I_4, the positions are shifted back, in I_3 by 1, in I_4 by $1 * 2 = 2$, to gather the sums. The final sums of each warp are then accessible for threads `threadIdx & (kWarpSize - 1)== (kWarpSize - 1)`. The *exclusive* scan version can easily be retrieved by deducting the original value provided in each thread as shown in Listing 4.4 .

The remaining thing for the parallel reduction algorithm to do is to warpscan these values again, and return the final sum to all threads via shared memory.

4.2.2 Stream Compaction

Stream compaction is a parallel primitive used to removed unwanted elements from sparse data [48]. In the context of this thesis, stream compaction is primarily used in the haar cascade object detection step in Section 4.3 to continuously filter out positions in an image that do not contain desired objects. In general, there are a lot of parallel algorithms that produce data containing unwanted elements, and where it is necessary to *compact* the data prior to further processing steps [48]. A naive, sequential implementation as shown in Algorithm 4.2 outlines the basic functionality of stream compaction.

Code 4.4 MSL Warp Exclusive Scan

```
template<typename T>
device T warpScanExclusive(
uint threadIdx,
T iData,
volatile threadgroup T *sData) {
    return warpScanInclusive<T>(threadIdx, iData, sData) - iData
        ;
}
```

Algorithm 4.2 Naive Sequential Stream Compaction

```
1: procedure NAIVESTREAMCOMPACTION
2:     $j \leftarrow 0$
3:     for $i \leftarrow 1 : n$ do
4:         if $valid\ in[i]$ then
5:             $out[j] \leftarrow in[i]$
6:             $j++$
7:         end if
8:     end for
9: end procedure
```

In our efficient parallel implementation, we will make heavy use of the scan algorithms outlined in Section 4.2.1 and mainly refer to the work by Billeter et al. [48] with a few modifications adapted to the specifics of the iOS platform. Generally, we again apply the condition that the number of threads processing the current threadgroup must be larger than the number of elements to be compacted. However, in our methodology, we implemented the possibility that each block can individually compact elements into device memory using atomic operations.

The stream compaction algorithm consists of three steps:

- Retreive a flag for each element indicating whether it is to be included in the compacted array
- Retreive the number of elements that will be inserted into the compacted array
- Gather valid elements and write them to the compacted array

We will describe the implemented algorithm using a simple example as indicated in Figure 4.5 .

We will use an initial array of size $N = 8$ containing the letters $A - H$. In the first step, we create a temporary mask array assigning ones to array elements which are to be compacted, and zeros to elements which should be skipped. We then perform an *inclusive prefix scan* on the mask array. The last element of the scanned array then contains the number of elements to be processed. Thus, each thread that includes a valid flag in the mask array (e.g. threads with indices $t_i = 0{,}2{,}4{,}6$), writes its own value into the final array at the position indicated by the scanned array. Thread t_6 for example first checks whether its processing a to-be-compacted

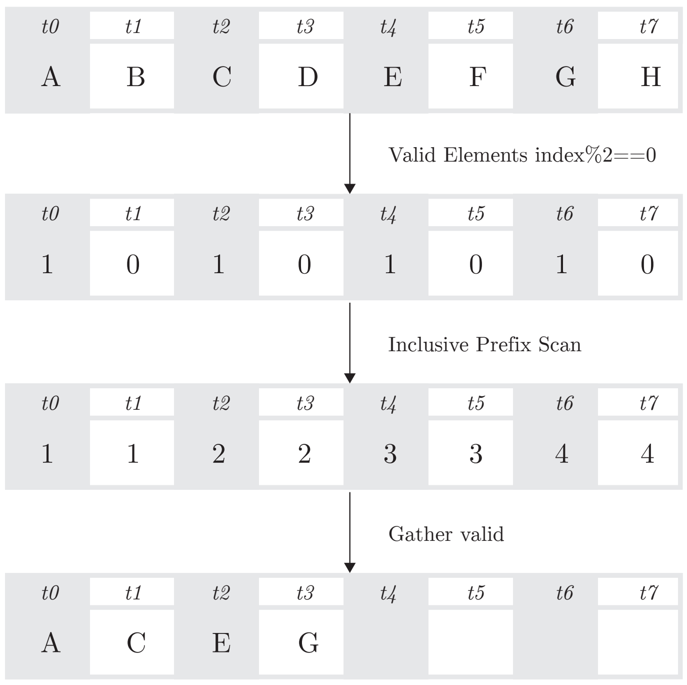


Figure 4.5: Stream Compaction Example

element (entry with index 6 in the mask array), and then retrieves the position to which to write to from the prefix scanned array (again entry with index 6, containing the value 4). The following Listing 4.5 illustrates this algorithm in MSL.

Each thread calls the function `streamCompactWriteOut` with two main arguments: first, the `threadPassFlag` indicating whether the value is to be written out, and second, the `threadElem` argument which contains the actual value (we only show the integer version of the function for simplicity). In the first step, each thread receives that scanned value at its position in the `incScan` variable. We then let the last thread with index `numThreads-1` set two values, `numPassed` and `outMaskOffset`. `numPassed` is itself a threadgroup variable and indicates the number of elements which will be present in the final array `vectorOut`. `outMaskOffset` indicates the start index of the first element in which values will be written. We use the MSL-native `atomic_fetch_add_explicit` which atomically adds the number of elements from our scan to a global device counter variable called `outMaskPosition` and returns the value the variable previously held. Using this approach, the function can be called simultaneously from several processing threadgroups with no appearing conflicts in terms of colliding positions. We then write out the `threadElem` value to a shared memory array at the index retrieved from the scan shifted by one (exclusive scan). The final step then consists of writing out the values now stored in shared memory to the final array `vectorOut` using the previously determined `outMaskOffset`.

Code 4.5 MSL Stream Compaction

```
device void streamCompactWriteOut(
        volatile threadgroup int *shmem,
        threadgroup int &numPassed,
        threadgroup int &outMaskOffset,
        uint threadIdx,
        int threadPassFlag,
        int threadElem,
        uint numThreads,
        device int *vectorOut,
        device atomic_int &outMaskPosition) {

    int incScan = scanInclusive<int>(threadIdx, threadPassFlag,
       shmem);

    threadgroup_barrier( mem_flags::mem_threadgroup );

    if (threadIdx == numThreads-1)
    {
        numPassed = incScan;
        outMaskOffset = atomic_fetch_add_explicit( &
           outMaskPosition, incScan, memory_order_relaxed );
    }

    if (threadPassFlag)
    {
        int excScan = incScan - threadPassFlag;
        shmem[excScan] = threadElem;
    }

    threadgroup_barrier( mem_flags::mem_threadgroup );

    if (threadIdx < numPassed)
    {
        vectorOut[outMaskOffset + threadIdx] = shmem[threadIdx];
    }

}
```

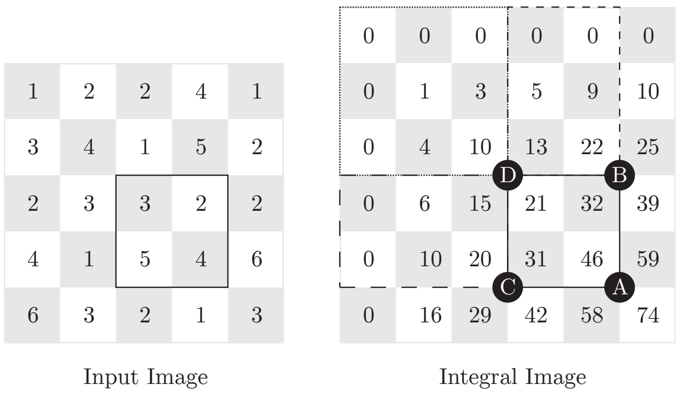
4.2.3 Integral Image Computation

Description & Purpose

In numerous algorithms, it is necessary to often compute the sum rectangular areas or windows of an image in order to calculate features at specific image positions. First introduced by Viola et Jones [28], the term *integral image* was coined representing an intermediary representation of an image where the location x,y contains the sum of the pixels above and to the left of x,y inclusive [28]:

$$ii(x,y) = \sum_{x' \leq x, y' \leq y} i(x', y') \tag{4.5}$$

Using the integral image, one can compute any rectangular sum in the original image using four references of the integral image. Given that for cascaded object detection a large number of these rectangular sums are required, this intermediary representation greatly speeds up the process of calculating image sums. Figure 4.6 shows an example image along with its (exclusive) integral image representation.



1	2	2	4	1
3	4	1	5	2
2	3	3	2	2
4	1	5	4	6
6	3	2	1	3

0	0	0	0	0	0
0	1	3	5	9	10
0	4	10	13	22	25
0	6	15	21	32	39
0	10	20	31	46	59
0	16	29	42	58	74

Figure 4.6: Integral Image Example

The sum of the marked rectangle area on the input image can now be easily computed using only four references to the integral image as follows

$$\text{rectangle sum} = A - B - C + D = 46 - 22 - 20 + 10 = 14 \tag{4.6}$$

Implementation

In our implementation, we referred to the ideas of Bilgic et al. [49] who implemented a parallel algorithm for computing the integral image on GPUs. The implementation consists of a number of operations:

1. Performing the row-wise scan operation on the entire image block-wise and writing block-sums to an auxiliary image
2. Performing the row-wise scan operation on the auxiliary image
3. Adding the values of the scanned auxiliary image to the scanned image

4. Transposing the scanned image
5. Performing operations 1-4 again and retrieve the integral image

This process is very similar to Section 4.2.1 for large arrays, the only difference being that the operations are performed row-wise on textures rather than on buffers. The MSL transpose kernel looks as follows (adapted from [49]):

Code 4.6 MSL Transposition

```
device void transpose(
                           texture2d<float, access::read> input,
                           texture2d<float, access::write> output,
                           uint2 blockDim,
                           uint2 blockIdx,
                           uint2 threadIdx,
                           threadgroup float4* sharedMemory
                           ) {

    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;

    if ((x < input.get_width()) && (y < input.get_height())) {
        sharedMemory[blockDim.x * threadIdx.y + threadIdx.x] =
           input.read(uint2(x,y));
    }

    threadgroup_barrier( mem_flags::mem_threadgroup );

    x = blockIdx.y*blockDim.x + threadIdx.x;
    y = blockIdx.x*blockDim.y + threadIdx.y;

    if ((x < input.get_width()) && (y < input.get_height())) {
        output.write(sharedMemory[blockDim.x * threadIdx.x +
           threadIdx.y], uint2(x,y));
    }

}
```

The main idea of this kernel is to first read a certain block of the image into shared memory, and then basically switch x and y coordinates in order to write onto the final image according to the values stored in shared memory.

4.2.4 Similarity Transformation

Within the implementation of regression-based facial feature detection, we iteratively need to adjust previously calculated shape features to an initial shape using a number of feature points. For this purpose, we need to estimate the rotation M, translation t and scaling c between two set of points x_i and y_i in two dimensional space. This is referred to as *similarity* *transformation*, and we follow the approach of Umeyama [50] by minimizing

$$e^2(R,t,c) = \frac{1}{n}\sum_{i=1}^{n}||y_i - (cRx_i + t)||^2 \quad (4.7)$$

in a least-squares fashion. The authors of [50] propose the following solution (see the paper for derivation and proof). First, we compute the covariance matrix $\sum_{x,y}$ of the points.

$$\sum_{x,y} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mu_y)(x_i - \mu_x)^T \quad (4.8)$$

where

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i \quad (4.9)$$

and

$$\mu_y = \frac{1}{n}\sum_{i=1}^{n} y_i \quad (4.10)$$

are the mean vectors of x and y. These operations are easy to parallelize given the scan algorithm in Section 4.2.1 . Then, assuming the singular value decomposition of $\sum_{x,y}$ is

$$\text{SVD}(\sum_{x,y}) = UDV^T \quad (4.11)$$

the authors derive that [50]

$$R = USV^T \quad (4.12)$$

$$t = \mu_y - cR\mu_x \quad (4.13)$$

and

$$c = \frac{1}{\sigma_x^2}\text{tr}(DS) \quad (4.14)$$

S must be chosen as

$$S = \begin{cases} \text{diag}(1,1), & \text{if } \det(\sum_{x,y} \leq 0) \\ \text{diag}(1,-1), & \text{if } \det(\sum_{x,y} < 0 \end{cases} \quad (4.15)$$

and σ_x^2 is the variance around the points in x. tr describes the trace of a matrix and is defined as

$$tr(A) = \sum_{i=1}^{n} a_{ii} \quad (4.16)$$

With these definitions, we can now easily define our shader computing the similarity transform given two point pairs as inputs. Given that there is no available library for the computation of the singular value decomposition for the MSL, which is basically a transformation of correlated variables into a set of uncorrelated ones using a set of orthonormal matrices (U,V) and a diagonal one (S) [51], we wrote a separate function computing the values U, S and V. Even though parallel approaches for the SVD computation have been proposed [52], we found that in our use case where the SVD input is a 2×2 matrix due to the 2-dimensional nature of our input points, a single thread performed best on computing the SVD. Thus, we utilize all available threads only until the final computation of the covariance matrix $\sum_{x,y}$ as well as the variance σ_x^2. We then return all threads with threadids larger than zero and let the remaining thread handle the remaining operations and the writing of rotation matrix and translation vector to device memory. For the SVD computation of our 2×2-matrix, we use the approach of [53] assuming that any 2×2-matrix can be decomposed into two rotations, and a nonuniform scale [54].

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{bmatrix} \begin{bmatrix} w_1 & 0 \\ 0 & w_2 \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma \\ -\sin\gamma & \cos\gamma \end{bmatrix} \tag{4.17}$$

We can then simply solve for w_1, w_2, β and γ using the resulting four equations [53]:

$$\begin{aligned} A &= w_1 \cos\beta \cos\gamma - w_2 \sin\beta \sin\gamma \\ B &= w_1 \cos\beta \sin\gamma + w_2 \sin\beta \cos\gamma \\ C &= -w_1 \sin\beta \cos\gamma - w_2 \cos\beta \sin\gamma \\ D &= -w_1 \sin\beta \sin\gamma + w_2 \cos\beta \cos\gamma \end{aligned} \tag{4.18}$$

4.2.5 Graph Partitioning & Grouping

In the context of object detection, there is often the problem that multiple bounding rectangles of desired objects are detected on pixels that are close to each other due to insensitivities to small changes in translation and scale [28]. Given that often only a single bounding rectangle is desired, it is necessary to *group rectangles* using a specific methodology. In, this Section, we will approach this problem in the same fashion as in [28], i.e. regarding rectangles whose bounding regions overlap to belong to the same object. The used function is illustrated in Listing 4.7 using a simple detection window structure containing position, width and height of the respective detection.

Code 4.7 MSL Overlapping Region Detection

```
bool isOverlapping(thread DetectionWindow w1, thread
   DetectionWindow w2) {
    return w1.x < w2.width+w2.x && w1.width+w1.x > w2.x &&
    w1.y < w2.height+w2.y && w1.height+w1.y > w2.y;
}
```

Overall, the problem is visualized in Figure 4.7 . In this example, a total number of seven detection windows are found in the image with assigned indices w_0-w_6. In our algorithm, we use the fact that multiple detections occur in the same image area as a means of verification that the object is correctly detected. When translating the problem into a graph, we simply define that two detection windows are neighbors if their regions overlap. We then specify a minimum number of neighbors in order for a detection to be marked as correct.

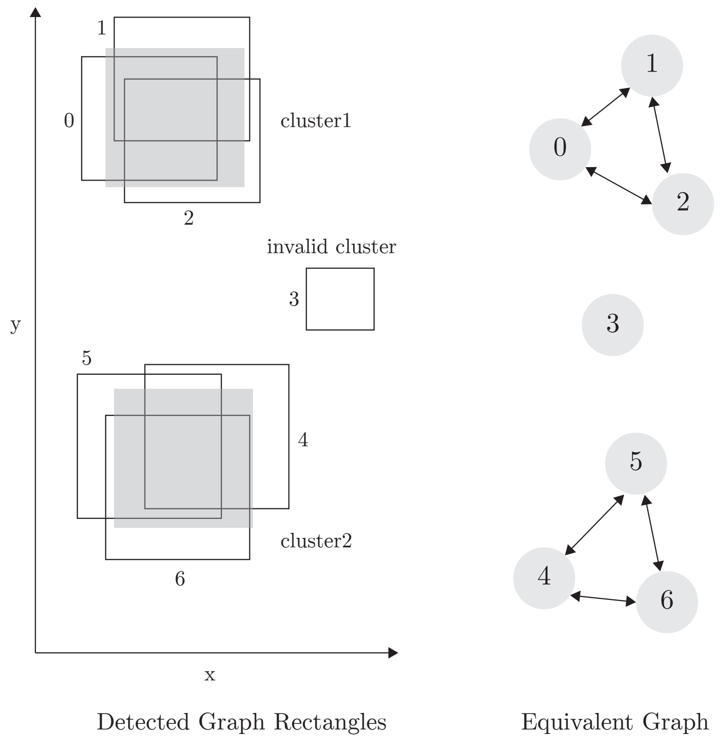


Figure 4.7: Rectangle Grouping Problem and its graph equivalent

In Figure 4.7 , one detection in w_3 does not have any neighbors and is thus considered to be a false detection in the subsequent computations. The remaining rectangles can then be clustered into two distinctive detections or *clusters*, where the final detection windows are computed using the average of neighboring rectangles. Therefore, the problem is equivalent to partitioning a graph using the overlapping region criterium.

Our algorithm to perform this operation on the mobile phone's GPU consists of three stages:

1. Build the Adjacency List
2. Group Rectangles
3. Gather Rectangles and build average clusters

In step one, we build intermediary buffers to gather neighbors of each rectangle or node. In step two, we build the clusters using the intermediary results from step one, and then build the average rectangles of each cluster in step three. These steps are described in more detail in the following Sections.

Adjacency List

Similar to [55], we will represent adjacencies in list form rather than the commonly used matrix form in order to avoid the often sparse nature of these matrices. The output of our first compute shader are three arrays: the adjacency list, and list of offsets on where to find adjacency information regarding each node in the adjacency list, and an array containing the number of neighbors of each node. The structure of these arrays is illustrated in Figure 4.8 .

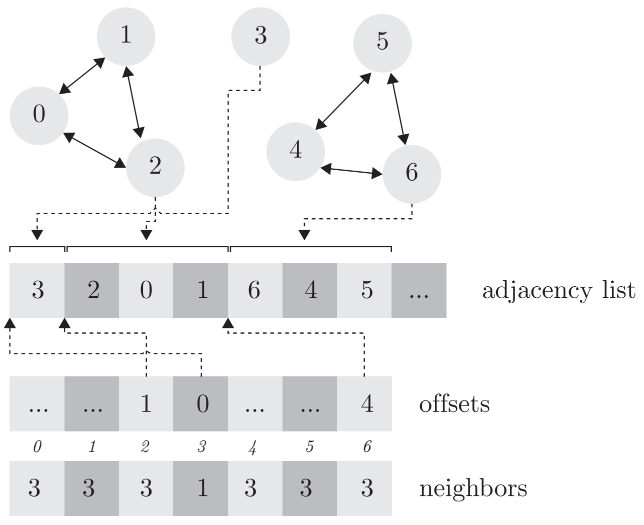


Figure 4.8: Rectangle Adjacency List, Offsets and Neighbors

We developed a novel algorithm to compute these arrays given that existing algorithms on GPUs are often tailored to a very large number of nodes in a graph, while in the typical use case of object detection we can never exceed the number of pixels in an image - and the number of actual detections is usually substantially lower than that theoretical number.

A sketch of the algorithm is shown in Algorithm 4.3 . In general, we issue one threadgroup per detected window or node with the maximum available number of threads on the platform `nThreads`. We then gather the rectangle `w_1` from our `windows` buffer which forms the basis for all threads processing this window. We then let all threads process all other windows by dividing the available number of windows by the available number of threads, and thus let a single thread process multiple windows in a loop. Note one needs to account for the case the remainder of this division is unequal to zero. We then compare our current window `w_1` with the next window in each loop `w_2` and set a flag `isInRegion` depending on whether the areas overlap. In each iteration of the loop, we then stream compact (see Section 4.2.2) the detected indices using the `isInRegion` flag. In each iteration, we store the current number of compacted windows using shared memory and use this value as an offset for the next stream compaction operation in the next loop iteration - an example is illustrated in Figure 4.9 .

In this fashion, we continuously store the neighbor indices of the current detection window in shared memory. Even though this method is extremely fast, it suffers from the limitation of maximum threadgroup memory. Using `uint32` types as indices with four bytes for each neighboring index, this enables a maximum neighborcount of roughly $n_{\max} = 4000$ given the threadgroup memory limitations of 16 kilobytes[2], minus threadgroup allocations required for other intermediate computations. In our tests we found that on average we were detecting a maximum number of windows of around 300-1500, so the memory requirements were always

[2]See https://developer.apple.com/metal/limits/

Algorithm 4.3 Parallel Adjacency List Building

```
1: procedure parallelAdjacencyList
2:     $w_1 \leftarrow$ windows[blockIdx]
3:     numChunks $\leftarrow$ numberOfWindows/nThreads
4:     neighborCount $\leftarrow 0$
5:     for $i \leftarrow 0$ : numChunks do
6:         isInRegion $\leftarrow 0$
7:         idx $\leftarrow i \times$ nThreads + threadIdx
8:         if $idx$ < numberOfWindows then
9:             $w_2 \leftarrow$ windows[idx]
10:            isInRegion $\leftarrow$ isOverlapping($w_1$,$w_2$)
11:            if isInRegion then
12:                neighborCount $\leftarrow$ neighborCount + 1
13:            end if
14:        end if
15:        streamCompactIntoSharedMemory(*sharedMemory*, isInRegion, idx)
16:    end for
17:    neighborCount $\leftarrow$ subReduceAdd(neighborCount)
18:    if threadIdx = 0 then
19:        neighbors[blockIdx] $\leftarrow$ neighborCount
20:        *threadgroup* offset $\leftarrow$ atomicFetchAddFromGlobalCounter(neighborCount)
21:        offsets[blockIdx] $\leftarrow$ offset
22:    end if
23:    *threadgroupBarrier*
24:    for $i \leftarrow$ threadIdx : neighborCount do
25:        adjacencies[offset + $i$] $\leftarrow$ sharedMemory[$i$]
26:    end for
27: end procedure
```

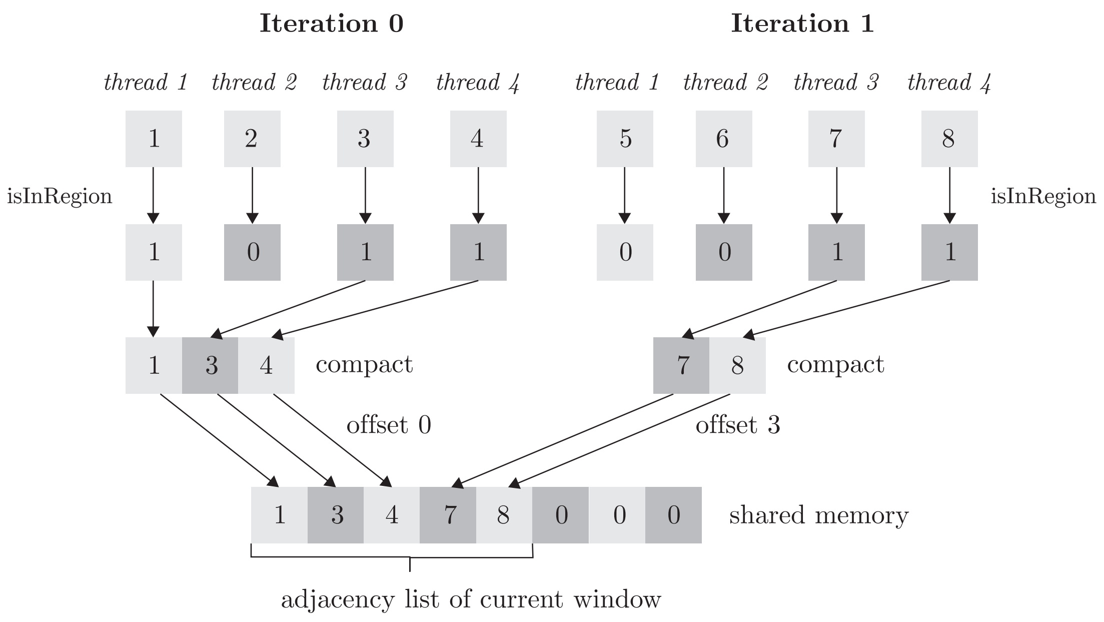


Figure 4.9: Adjacency List Stream Compaction Example

sufficient for our pipeline. Nevertheless, we limited the maximum number of detected windows to 1500 in order to avoid memory overflows or accidental access of random memory.

Simultaneously, we count the number of valid neighbors found in each thread and then, upon exiting the first loop, subreduce (see Section 4.2.1) that value using addition in order to get the final number of neighbors for the current detection window. In the next step, the first thread with `threadIdx = 0` stores the number of neighbors using the `blockIdx` id of the current detection window.

Given that the order of execution of threadgroups is not fixed (see Section 2.1.2), we need to account for the possibility that `blockIdx` with larger values are issued first. For this, we use the `atomic_fetch_add_explicit` (see Section 2.2.3) function to atomically get a value for an arbitrary offset from a global, device-based variable. One the one hand, we retrieve the current value of that variable, which forms the index for our first entry of the adjacency list of the current detection window, while on the other hand we increase that variable using the number of found neighbors. Therefore, other blocks and their written adjacency lists can never overlap with the current adjacency list of the detection window. The only remaining thing to do is then to write out the values stored in `sharedMemory` from our stream compaction operations using the offset from the atomic fetch.

Rectangle Grouping

The previous operation resulted in three different buffers: the neighbor-count buffer, i.e. the number of detected neighbors of each window, the offset buffer, specifying at which position to find information about neighbors in the adjacency buffer, and the adjacency buffer itself.

In the next step, we now need to cluster all detection windows according to their neighbors. In our implementation, we disregard windows with a neighborcount lower than three, which has shown to yield optimum results. The desired output of this operation is firstly, the detected number of clusters, and secondly, an array containing to which cluster each detection window belongs. In case a window has a lower number of neighbors than the set threshold, we set the respective array value negative to indicate that this rectangle should not be considered in further processing steps. In contrast to the previous step, we are only using a single threadgroup for this stage. All threads in this threadgroup continuously process a single cluster until no more rectangles are found within that cluster, and then increase the number of clusters and repeat this step. This is performed until no more rectangles are found to be processed.

We will introduce two helper functions in order to split up the complexity of the shader. First, Algorithm 4.4 illustrates how a single thread can fetch a next valid position of a detection window to process: we iteratively increase a counter variable in the threadgroup address space until this counter exceeds the number of detected windows.

Algorithm 4.4 Rectangle Grouping Fetch Position

```
procedure fetchPosition(&position)
    while counter < numberOfWindows do
        if not visited[counter] and neighbors[counter] $\geq$ minNeighbors then
            position $\leftarrow$ counter
            counter $\leftarrow$ counter + 1
            return true
        else
            counter $\leftarrow$ counter + 1
        end if
        return false
    end while
end procedure
```

In each iteration, we check the node at the counter position on two conditions: whether it has not been visited by our algorithm before and whether the minimum neighbors requirement is fulfilled. If both conditions are met, the function returns that it has successfully found a new position to process along with the index of this window.

Algorithm 4.5 shows how the found position of Algorithm 4.4 is then processed.

We apply a breadth-first approach similar to [55] using two arrays: visited and frontier. The visited array simply indicates whether a node has been visited before, and is also used in Algorithm 4.4 to determine the next position. The frontier array shows which nodes are currently being processed. We use position found in Algorithm 4.4 , i.e. the first unvisited node, to mark all nodes that are being visited throughout the search. We then start the processing algorithm by setting an element in the frontier array to the current position (which serves as a marker), and use this element as the starting point for our breadth-first search. Each thread in the threadgroup processes a distinct (or multiple distinct) window, and sets the visited flag as well as the belonging cluster. Next, we find the neighbors of the currently processed window through the adjacency list and mark their indices in the frontier array. If an unprocessed neighbor is found, the `continueToProcess` flag is atomically set to true, and the outer loop continues. This happens until no more nodes are found to process.

Algorithm 4.5 Rectangle Grouping Process Position

```
1: procedure processPosition
2:     while shouldContinue do
3:         if threadIdx = 0 then
4:             storeExplicit(continueToProcess, false)
5:         end if
6:         threadgroupBarrier
7:         for $i \leftarrow$ threadIdx : numWindows, $i = i +$ nThreads do
8:             if frontier[$i$] = $position$ and not visitied[$i$] then
9:                 visited[$i$] $\leftarrow$ true
10:                clusters[$i$] $\leftarrow$ currentCluster
11:                for $j \leftarrow$ offset[$i$] : offset[$i$] + neighbors[$i$] do
12:                    frontier[adjacencies[$j$]] $\leftarrow$ $position$
13:                    fetchOrExplicit(continueToProcess, true)
14:                end for
15:            end if
16:        end for
17:        threadgroupBarrier
18:        if threadIdx = 0 then
19:            shouldContinue $\leftarrow$ loadExplicit(continueToProcess)
20:        end if
21:        threadgroupBarrier
22:        if not shouldContinue then
23:            break
24:        end if
25:    end while
26: end procedure
```

Algorithm 4.6 shows how the two Algorithms 4.4 and 4.5 are being called.

Algorithm 4.6 Rectangle Grouping

```
1: procedure groupRectangles
2:     while shouldContinue do
3:         if threadIdx = 0 then
4:             hasNextPosition ← fetchNextPosition(position)
5:             if hasNextPosition then
6:                 shouldContinue ← false
7:             else
8:                 shouldContinue ← true
9:                 frontier[position] ← position
10:                clusterCount ← clusterCount + 1
11:            end if
12:        end if
13:        threadgroupBarrier
14:        if not shouldContinue then
15:            return
16:        end if
17:        processPosition(position, clusterCount)
18:        if threadIdx = 0 then
19:            shouldContinue ← true
20:        end if
21:    end while
22: end procedure
```

The first thread in the threadgroup determines the next free position to process using the `fetchNextPosition` function, and marks the position in the frontier array so the `groupRectangles` procedure can use this node as start for further searching. In addition, a device-based variable `clusterCount` increases upon every iteration of the outer while-loop.

We then process all nodes that are reachable from this position, and finally return the function once all nodes have successfully been processed.

Rectangle Gathering

Using the `clusterCount` variable, we can now issue $n =$`clusterCount` threadgroups to gather the average desired rectangle or cluster. Each threadgroup processes a different cluster, and all threads in a single threadgroup are used to sub-reduce the actual detection windows (i.e. x and y position as well as width and height) and return the average rectangles for each cluster by dividing the summed-values by the total number of rectangles in the cluster.

4.3 Cascaded Classifier Face Detection

4.3.1 Overview

This Section describes our implementation of the object detection framework by Viola et. Jones [28]. It is still regarded as one of the most important contributions to this field and has only marginally changed since it was originally published [3]. In general, our implemented algorithm consists of the following steps:

- Creation of the object classifier structures in GPU memory upon initialization
- Preprocessing of the to-be-analysed image per frame
- Applying the object classification per frame
- Grouping the detected objects per frame

For details regarding the general object detection methodology, we refer the reader to Section 3.3.2 . We use a previously trained list of cascaded classifiers which is loaded into memory at runtime - see Section 4.3.5 - to detect a bounding rectangle of a face in the image acquired from the smartphone's camera.

Note that we do not use rotated Haar-like features [56] for obvious performance reasons (multiple rotated integral image computations necessary) and thus sacrifice face detections where the user's face is strongly tiled in either direction. This would be a potential improvement point for further development of our facial features detection pipeline.

4.3.2 Face Classifiers & Data Structure

Before any processing steps can be applied, the previously trained list of classifiers needs to be stored efficiently in GPU memory. In the method by [28], we use a cascade of classifiers. A cascade of classifiers is trained for a fixed detection window size (24×24 pixels in our case).

Each of the steps in the cascade is referred to as stage, where each stage either detects or does not detect the face - in case a face is detected, further stages start their processing until the last stage is reached. Our classifier (see Section 4.3.5) has the structure as shown in Figure 4.10 .

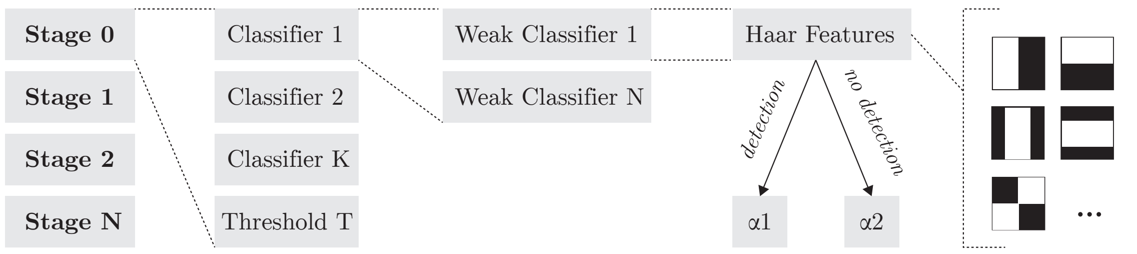


Figure 4.10: Object Detection Classifiers, adapted from [3]

A single stage is composed of K *classifiers* and a stage threshold T. Each *classifiers* then contains a set of N so-called *weak-classifiers*. *weak-classifiers* can individually not make any detection on object level [3], but only respond according to their respective filters, hence their naming. *Weak-classifiers* consist of a set of a tree or stump of *haar features*. A *haar feature* simply specifies a rectangle to be laid over the image at a specific location, and returns the sum of all pixels in that rectangle. In Figure 4.10 , they are shown as their typical appearance: as black-and-white coloring scheme. Given a *weak-classifiers* consists of at least two *haar* *features*, the black part usually makes a positive contribution to the sum of pixels (i.e. is added), while the white part is deducted from the sum of pixels (i.e. subtracted). *weak-classifiers* specify this relationship using weights, and the resulting value of a *weak-classifiers* is determined by the weighted value of the sum of pixels in the rectangle of the individual *haar feature*. This summation is then compared to a threshold on *weak-classifier*-level: if the summed value is larger than the threshold, the *weak-classifier* returns value α_1, otherwise α_2. If a weak classifier contains a single set of haar features, it is referred to as stump-based. The other possibility, a tree-based weak classifier, is shown in Figure 4.11 .

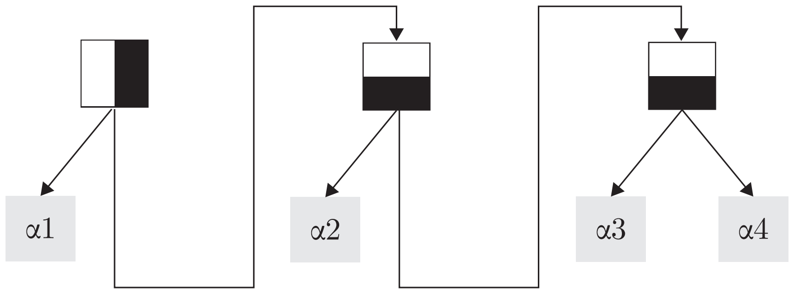


Figure 4.11: Tree based weak classifier

In this case, each haar feature can contain leaf-nodes, i.e. depending on the result of the haar feature and the comparison with a threshold, the next haar filter is called until the last haar filter is reached. For simplicity reasons and because our pre-trained classifier set did not contain any tree-based weak classifiers, please note that we will only describe the stump-based version of our implementation.

The existing classifiers are not well-suited to be stored directly in GPU memory, this is why we employ a layout scheme similar to [3]. This is shown in Figure 4.12 .

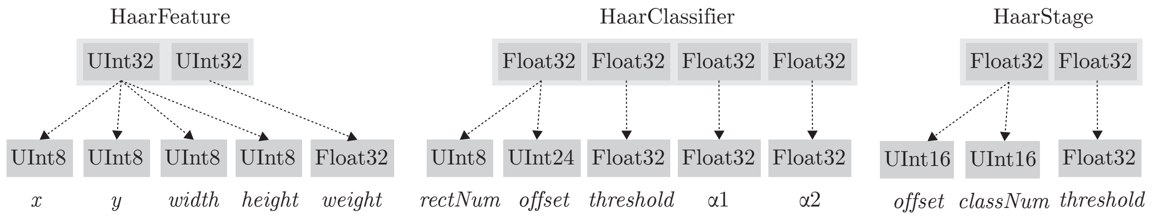


Figure 4.12: Feature, Classifier and Stage Encoding for Object Detection

We require three entities that need to be stored efficiently:

- Haar Features
- Classifiers
- Stages

Haar Features

A single haar feature consists of five values: x and y, i.e. the pixel offset of the rectangle within the detection window, width and height, i.e. the dimensions of the feature rectangle, and weight, i.e. the weight to use when computing the sum of a weak classifier. Two implementational constraints need to be mentioned at this point: created metal-based textures with a specific number on channels on the iOS platform only support a single data-type for the entire texture. Thus, one can not have one floating-point channel mixed with an integer-value channel. Second, iOS metal supports reading up to 16-bytes at once from either a buffer or texture, thus in order to optimize reads from device-memory it is optimal to pack all required structures into data-types of size 16,32, 64 or 128 in bits [3]. One exampled on how this is used is shown on the example of haar features in Figure 4.12 . We use a two-dimensional texture with two uint32 channels to store haar features. We then split the first channel, consisting of 4 bytes, into four different uint8 data types, while converting the second channel to a floating-point value. Code 4.8 shows on how this is achieved in the MSL.

Code 4.8 MSL Reading of Haar Features

```
uint2 values;
uint height = (uint) (values[0]>>24);
uint width  = (uint) ((values[0]&0xFFFFFF)>>16);
uint y =      (uint) ((values[0]&0xFFFF)>>8);
uint x =      (uint) ((values[0]&0xF));
float weight =  as_type<float>(values[1]);
```

Given the fixed window size of 24×24 pixels, we can safely assume that x, y as well as width and height of the haar feature will never exceed the uint8 data type with $2^8 = 256$ possible values. The first value, height, is easily gathered by byteshifting the value to the right by $32 - 8 = 24$ bits. This leaves the remaining 8 bits padded with zeros at positions $0 - 24$, thus returning a `uint8` data type. For the next values, we first need to mask the original value. The masking value for width is `0xFFFFFF`, which is 8 bits of zeros followed by 24 bits of ones. By performing a bitwise `AND` operation, we effectively clear out the first 8 bits of the original value. As the actual height value is stored from bits at positions $8-16$, we only need to bitshift by 16 positions to retrieve the original value. x and y are obtained in a similar fashion, only that the masking value is shorter, and that initial positions of the masking value are automatically padded with zeros upon the bitwise `AND` operation. The weight on the other hand is stored as floating-point type, thus the only thing to do is to reinterpret the texture data using the `as_type` operation from the Metal STL.

Classifiers

Classifiers are stored in a texture using four `float32` channels. While we use the last three channels directly for our thresholds, the first value is again split up in a similar fashion as it was the case of haar features. We use the first 8 bits to store the number of rectangles or haar features this classifier contains (which should never exceed 2^8), as well as the offset at which the first haar feature of this classifier is to be found. The corresponding Code 4.9 shows on the values are retrieved.

Given that both the haar feature texture and classifier texture are two-dimensional, we again use the same methodology to access x and y positions in a texture as shown in Code 4.10

Code 4.9 MSL Reading of Classifiers

```
float4 values;
uint numRects = as_type<uint>(values[0]) >> 24;
uint firstRectOffset = as_type<uint>(values[0]) & 0xFFFFF;
float threshold = values[1];
float left = values[2];
float right = values[3];
```

in order to avoid the necessity of storing two separate values.

Code 4.10 MSL Retreiving Coordinates from a single value

```
uint mask;
uint y = mask >> 16;
uint x = mask & 0xFFFF;
```

Stages

The last structure is a single stage. It basically consists of three different values: again, similar to classifiers, an offset value indicating on where the first classifier of this stage can be found in the corresponding texture, the number of classifiers of this stage, and the stage threshold. As shown in Figure 4.12 , we use the first value of the stage buffer and split it into two values containing offset and classifier number. The code for this operation is equivalent to Code 4.10 .

Figure 4.13 shows how these structs are layed out in memory.

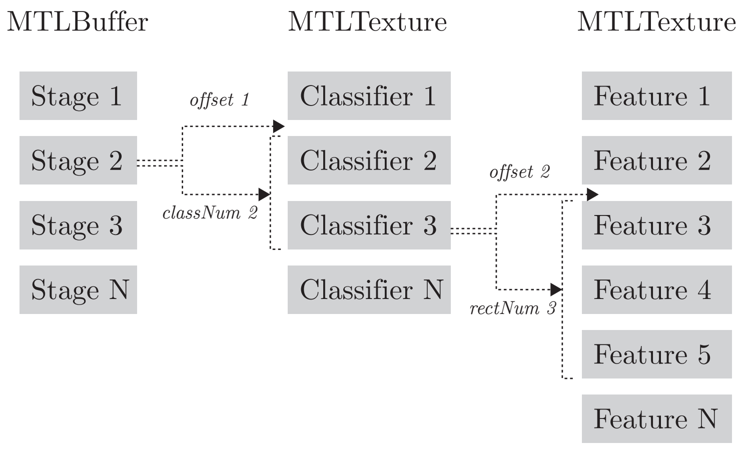


Figure 4.13: Visual Memory Layout of Stages, Features & Classifiers

Each stage points to a specific offset in the classifiers texture, while each classifier points to a specific offset in the haar features texture. Using this methodology and layout provides a very efficient in terms of required memory size to read from device-based memory.

4.3.3 Preprocessing & Scale Invariance

Given that the detection window of the used classifier file is fixed, there are two possible options of detecting faces on different scales. These options are shown in Figure 4.14 .

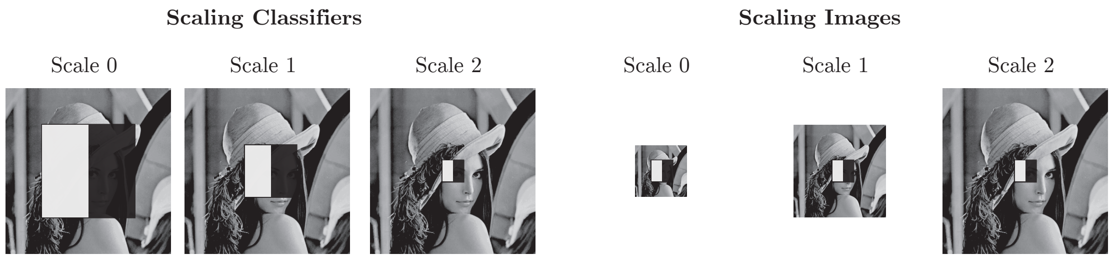


Figure 4.14: Scaling Options for Object Detection

One option is to to simply scale the image to various sizes (i.e. build an image pyramid) and then apply the classification procedure on each of those images. The second option is to scale the classifier, i.e. scale the detection window according to the current scale factor and thus the included haar features. The first option is considered the most precise, however, also the computationally most expensive as all integral images and other preprocessing steps need to be repeated for each of the scaled images. With the second option, all processing steps remain intact, however, when up- or downscaling the original window by a real value, the features do not map to the pixel grid anymore [3]. x, y, width and height of each haar feature are integer values, thus rounding coordinates to nearest pixels results in precision loss. The precision loss manifests itself in the weights of the haar features that are applied, and the usual way to mitigate this is by adjusting the haar feature weights according to the changed area of the detection window [3]. In our implementation, we opted for the second option due to obvious performance reasons. The biggest disadvantage is seen on very small scales, which should not affect our face detection because we generally set the minimum size of the desired face to be found in the image not to be too small.

In general, the preprocessing steps of the object detection can be summarized as follows:

- Convert to Grayscale
- Scale Image
- Integral Image
- Squared Integral Image
- Compute Variance Map

First, the cascade classification is trained and thus performed on grayscale images. As especially the computation of the first few stages is expensive in terms of frame time, we further scale the input image by a factor of 2 downward in order to avoid having to issue too many threadblocks on the first few stages. The computation of the integral image, as described in Section 4.2.3 is required for fast computation of haar features. The squared integral image is computed in the same way as the integral image except that all input values are squared before the sub-reduction operation and is needed for the computation of the variance map, which is described in the following.

Variance Map

In order to minimize for varying lighting conditions, the used classifiers in our implementation were trained on variance-normalized images, which can be obtained by dividing individual image pixel values by the standard deviation of all pixels in the image [3]. This value can be calculated for a detection window X of size $M \times N$as shown in Equation (4.19) [3].

$$\mathrm{Var}(X) = \frac{1}{MN} \sum_{i=0,j=0}^{M,N} X[i,j]^2 - (\frac{1}{MN} \sum_{i=0,j=0}^{M,N} X[i,j])^2 \tag{4.19}$$

Now it becomes clear why we need to compute the squared integral image in our preprocessing steps, namely the first term in Equation (4.19) . The second term, the mean, can then be easily obtained using the integral image. During the detection process, we achieve the effect of normalization by adjusting the classifier threshold as shown in Equation (4.20) . This also shows on how the adjustment for the previously mentioned rounding-precision problem is incorporated.

$$\mathrm{ClassifierThreshold}(X) = \mathrm{ClassifierThreshold}(X) \times \sqrt{\mathrm{Var}(X)} \times \mathrm{scaledArea}(X) \tag{4.20}$$

In our implementation, we store the $\mathrm{Var}(X)$ value for each detection window between intermediate processing shaders in order to avoid recomputation.

4.3.4 Object Classification

Overview

This Section describes the actual implementation of the face detection process. After read the classifier structure in device-memory and having computed the integral image and the variance map, we now need to check every location at (x,y) in the image at different scales on whether the respective detection window passes through all stages and their classifiers. A naive implementation of this algorithm is shown in Algorithm 4.7 .

To speed up our implementation, we have chosen the following criteria in order to determine the scale levels of our classifiers. We set the minimum size of the detection window to be at least `min(width,height)` of our scaled input image as we are only interested in detection face closely placed in front of the camera. On the other hand, we set the maximum size of the detection window to be `max(width,height)`, which forms the largest possible rectangle that can theoretically be fit into the image. We set the intermediary scale levels according to the original paper [28] to $s_{\mathrm{level}} = 1.2^{\mathrm{level}}$.

The above algorithm is, however, not well suited for the parallel-nature of GPUs. Figure 4.15 shows the required operations visually in more detail.

First, each stage is processed individually and depending on whether the previous stage

Algorithm 4.7 Naive Object Detection Algorithm

```
procedure GATHERFACES
    faceList ← []
    for $scale$ in all scales do
        for $x$ in width do
            for $y$ in height do
                for $s$ in stages do
                    stageSum ← 0
                    for $c$ in s.classifiers do
                        stageSum ← stageSum + c.calculateReponse($x$,$y$)
                    end for
                    if stageSum < s.threshold then
                        faceList.append(currentWindow)
                    end if
                end for
            end for
        end for
    end for
    groupRectangles(faceList)
end procedure
```

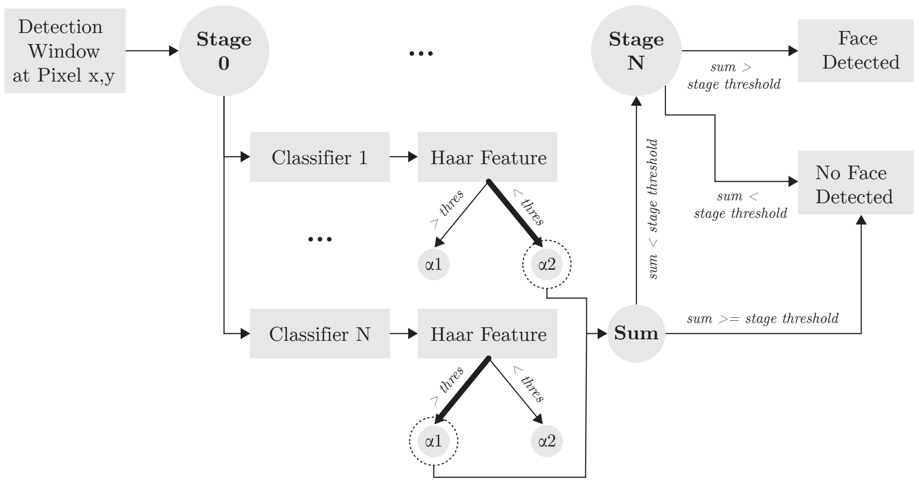


Figure 4.15: Face Detection Process for a single Scale

returned a positive response. Within each stage, each classifier independently calculates its intermediate response from its haar features. Each haar feature in turn uses the passed integral image to compute the sums of its filter rectangles and then compares them to the stored classifier threshold. In case the summed responses from the haar rectangles are larger then the threshold, the classifier returns the value α_1 to the stage, α_2 otherwise. All the values from the classifiers are then summed and compared against the threshold stored at stage level. In case the stage threshold is smaller than the summed classifier values, the next stage is processed. Otherwise the current position is marked as a non-face region. The upcoming Sections describe this process in more detail.

Haar Feature Response

The first important procedure of the detection process is the computation of haar feature responses. Figure 4.16 shows how an individual classifier obtains its haar feature response.

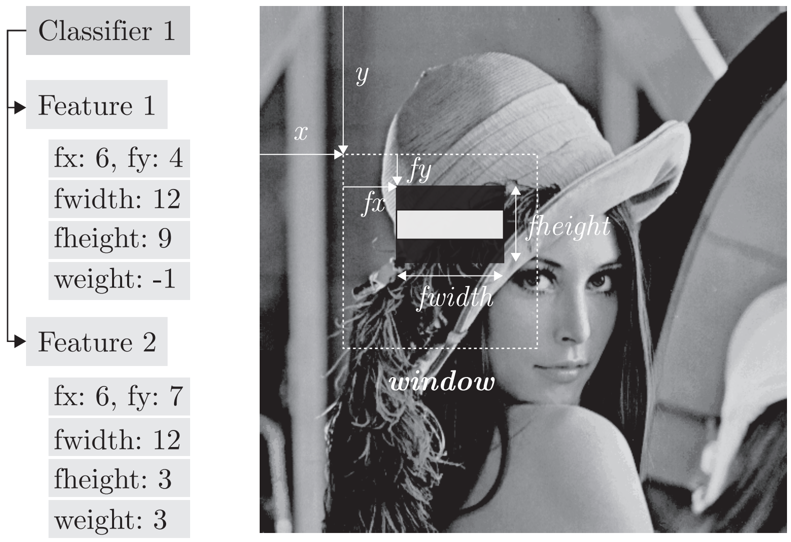


Figure 4.16: Individual Classifier Detection

Processing

- Pixel-Parallel from Texture
- Pixel-Parallel from Buffer
- Stage-Parallel from Buffer

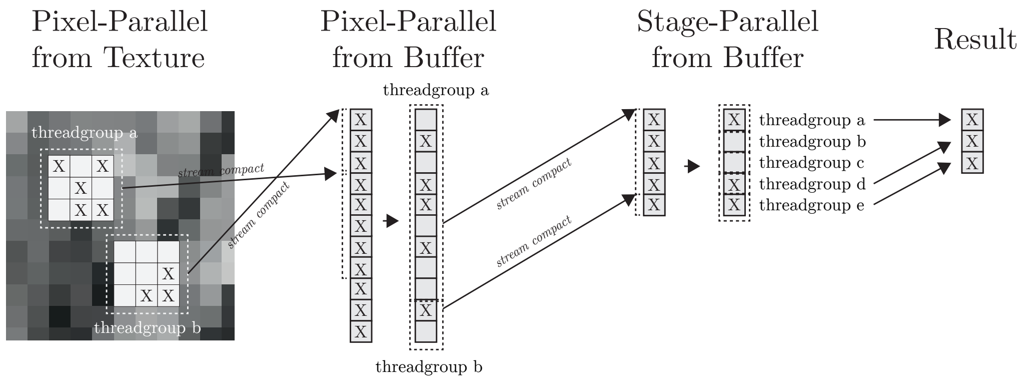


Figure 4.17: Face Detection Process Implementation for a single Scale

4.3.5 Cascade Dataset

We use the stump-based adaboost frontal face detector of 24x24 pixel size created by Rainer Lienhart for the OpenCV library [57] [58]. This classifier is typically stored as an XML file

Table 4.1: Trained Classifiers Overview

Stage	Number of Classifiers
1	9
2	16
3	27
4	32
5	52
6	53
7	62
..	...
24	211
25	200

of around 1MB in size and this classifiers as well as similar ones created for OpenCV have become a de facto standard for general-purpose frontal face detection [3]. This detector was trained using Discrete Adaboost as described in [59], which is a powerful machine learning algorithm capable of learning strong classifiers based on a large set of weak classifiers by continuously re-weighting the training samples [29]. For details on the implementation of this learning algorithm we refer the reader to [59].

Table 4.1 gives an overview over a number of stages in that detector along with the number of classifiers in each stage.

Before reading in the respective face detector classifier file, we convert it to JSON for simpler parsing and reading. This then serves as data entry-point for filling the data structures described in Section 4.3.2 .

4.3.6 Grouping

Figure 4.18 gives an example of the typical output of our face detection processor. As processed detection windows which are close to each other often produce positive responses on different scales, we need to merge these bounding rectangles in order to retrieve a single bounding rectangle of the face.

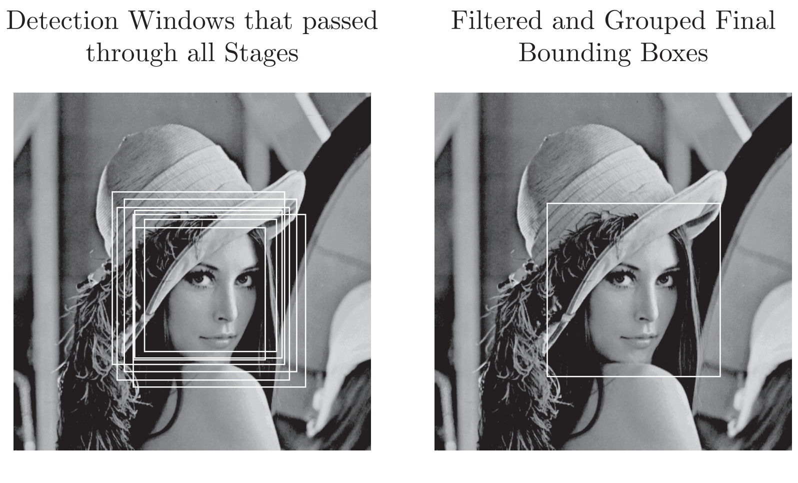

Detection Windows that passed through all Stages
Filtered and Grouped Final Bounding Boxes

Figure 4.18: Face Detection Grouping

We use our rectangle grouping algorithm as described in Section 4.2.5 to group these detection windows on the criterium that they overlap as per the original paper of Viola et. Jones [28].

4.4 Facial Feature Detection using Regression Trees

4.4.1 Overview

As stated in Section 3 , we use the algorithm of [21] in our implementation for detection the desired feature points. Now that the implementation regarding the retrieval of the bounding box of the respective face in the image has been described in the previous section, this Section shows how the facial feature points or face landmarks are found within that rectangle. We use the 68-point based face landmark annotation retrieved from the iBUG 300-W face landmark dataset [4] as shown in Figure 4.19 .

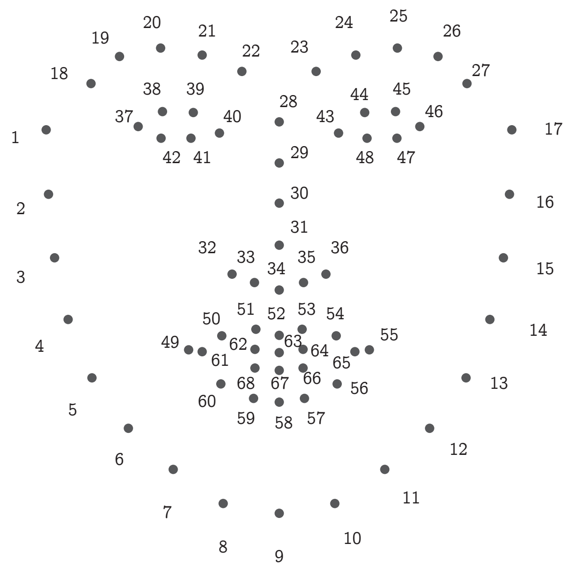


Figure 4.19: Facial Feature Detection Landmarks used in Implementation

For information on training the values used in this algorithm we refer the reader to Section 4.4.2 . The used landmarks describe not only the general shape of the face, but also eye contours, nose as well as mouth shape.

Similar to the process of object detection, a series of cascades is employed again in order to detect the facial landmarks. Let $x_i \in \mathbb{R}^2$ be the (x,y) coordinates of the i-th landmark in the current frame I. Then the vector $S = (x_1^T, x_2^T, \ldots, x_M^T)$ denotes the coordinates of all facial landmarks in I [21] - in the following we will refer to S as shape.

In each of the cascades t, a regressor r_t being an ensemble of regression trees - also referred to as random forest - is used to determine an incremental update to the current shape S^t form, starting from the mean shape S_{mean} of all shapes used for training as shown in Equation (4.21) [21].

$$S^{t+1} = S^t + r_t(I, S^t) \tag{4.21}$$

Thus, $S^1 = S_{\text{mean}} + r_t(I, S_{\text{mean}})$ in the very first cascade. The regressor of each cascade r_t is a function of the previous shape S^t and the grayscaled image I. As stated before, the regressor consists of a series of regression trees, which each of these trees being trained to minimize the alignment error in the least squares sense. Figure 4.20 shows how the regression trees are evaluated in each cascade.

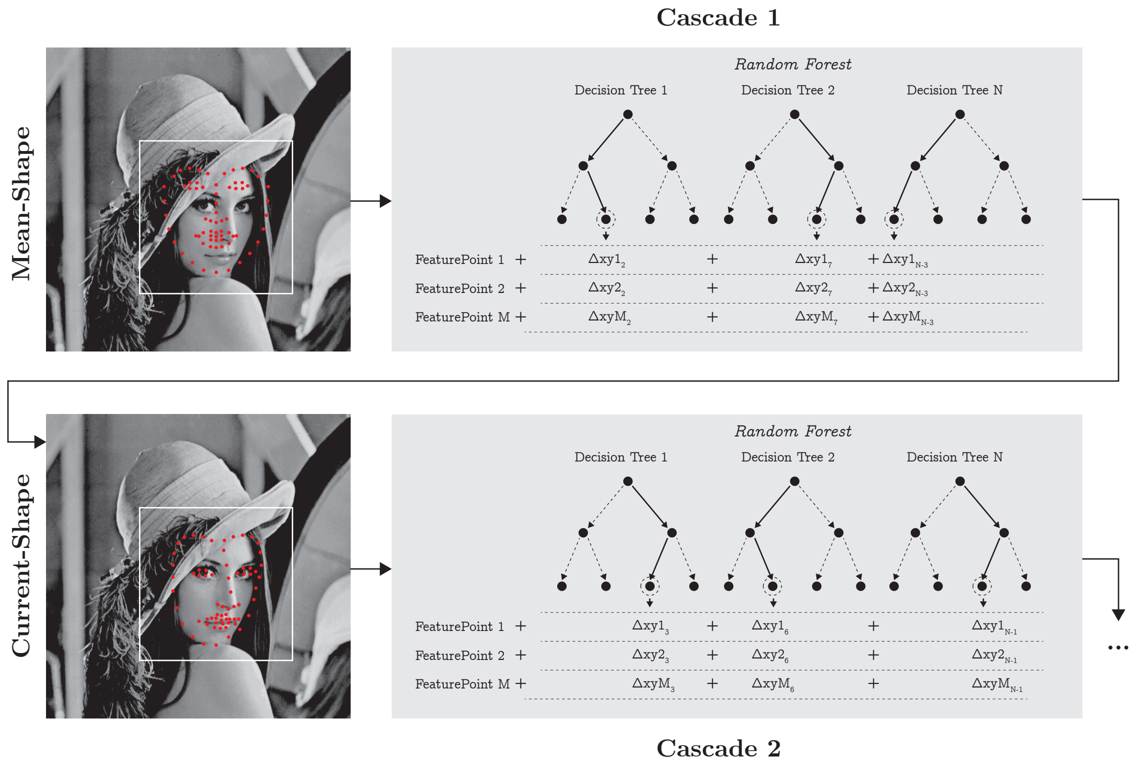

Cascade 1
Mean-Shape
Random Forest
Decision Tree 1
Decision Tree 2
Decision Tree N
FeaturePoint 1 + Δxy1_2 + Δxy1_7 + Δxy1_N-3
FeaturePoint 2 + Δxy2_2 + Δxy2_7 + Δxy2_N-3
FeaturePoint M + ΔxyM_2 + ΔxyM_7 + ΔxyM_N-3
Current-Shape
Random Forest
Decision Tree 1
Decision Tree 2
Decision Tree N
FeaturePoint 1 + Δxy1_3 + Δxy1_6 + Δxy1_N-1
FeaturePoint 2 + Δxy2_3 + Δxy2_6 + Δxy2_N-1
FeaturePoint M + ΔxyM_3 + ΔxyM_6 + ΔxyM_N-1
...
Cascade 2

Figure 4.20: Facial Feature Detection using Regression Trees Overview

Each decision tree in the random forest of each cascade is evaluated independently. Starting at the root node, we evaluate splits on every node and then traverse the tree until a terminal leaf node is reached. Each leaf node in every decision tree contains a list of a pair of Δx and Δy values for each feature point. These values are then added to the shape passed on from earlier cascades. In this fashion, the face landmarks in S^t are updated to S_{t+1} and the resulting shape theoretically moves closer to the actual ground truth. Figure 4.21 illustrates this process using a simple example.

When storing the indices of splits in consecutive order as shown in Figure 4.21 , we can easily retreive the index j of the next split using $j_{s+1} = 2 * j + 1$ in case the split directs to the left branch, and $j_{s+1} = 2 * j + 2$ using the right branch. After subtracting the number of splits from j at the resulting terminal node, we retreive the index of the leaf value that is added to the current shape.

A split in the decision tree is formally a triple (p_1, p_2, t) where p_1 and p_2 are the coordinates whose difference is a feature $f = p_1 - p_2$, and t is a threshold this feature is compared against. If $f > t$, then we follow the left branch of the current node, otherwise we follow the right branch. Thus, the decision at each node is based on thresholding the difference of intensity values at a pair of pixels [21]. This is methodology was chosen due to its relative insensitivity to changes in lighting.

Rather than storing coordinates in p_1 and p_2, we store indices to an array of offsets from the facial landmarks in the mean shape S_{mean}. This poses another problem: during training,

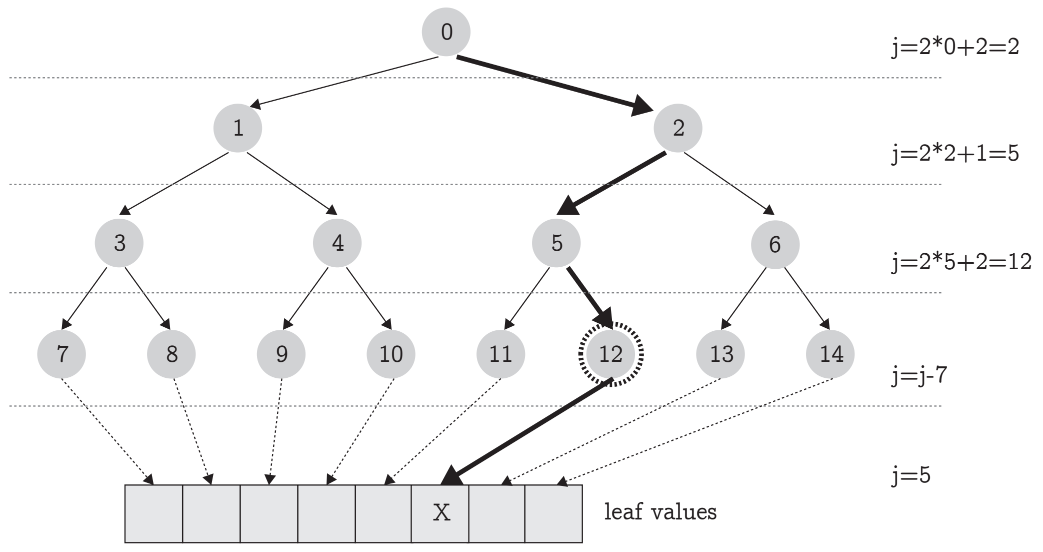
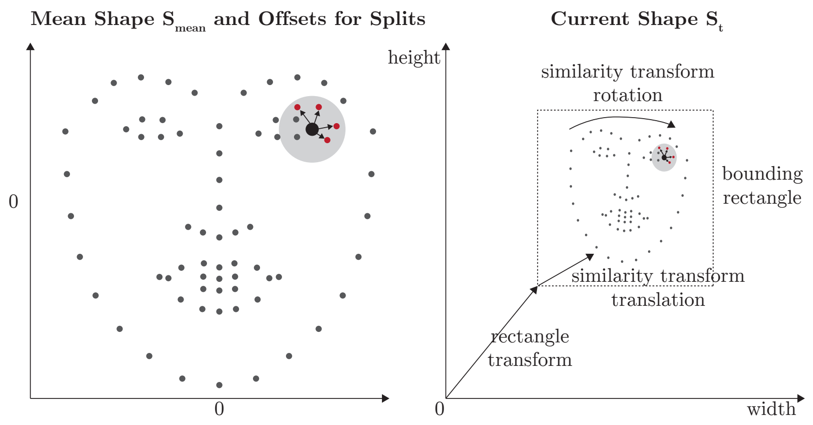


Figure 4.21: Single Decision Tree Example

we use random offsets from facial landmarks of S_{mean}, which is always stored in a normalized coordinate system, i.e the center of all coordinates in x and y direction is positioned at (0,0). Furthermore, given that the shape S_t changes upon every iteration in the cascade, we need to account for those changes when computing the values (p_1, p_2) at each split node. Figure 4.22 illustrates how this is performed visually.



Mean Shape S_mean and Offsets for Splits
Current Shape S_t
height
0
0
0
width
similarity transform rotation
bounding rectangle
similarity transform translation
rectangle transform

Figure 4.22: Feature Pixel Values Offset Transformation

Each split references two offsets from a specific feature point. The offset points are indicated by red dots in Figure 4.22 . In order to transfer these offsets into the current image, two transformations are necessary. First, we need to both rotate and scale the offsets from a landmark according to the estimated rotation and translation from $S_{\text{mean}} \rightarrow S_t$. Second, we need to transform the offsets from the normalized coordinate space of the mean shape into the coordinate space of the image, using a rectangle transform. The first transformation is achieved using the similarity transform as described in Section 4.2.4 . The latter is simply translating the offsets by the top-left corner of the bounding rectangle, and scaling by width and height of the rectangle. Denoting the translation of the similarity transform $S_{\text{mean}} \rightarrow S_t$ t_{s} (vector of size two), and the rotation matrix M_{s} (matrix of size 2×2), we can find the

Table 4.2: Shape Predictor Arrays after Training

Type	Number of values	Value Type	Bytes
Mean Shape	68×2	Float	544
Anchors	15×500	Int	30.000
Deltas	$15 \times 500 \times 2$	Float	60.000
Splits	$15 \times 500 \times 15 \times 3$	Int/Float	1.350.000
Leaf Values	$15 \times 500 \times 16 \times 68 \times 2$	Float	65.280.000

desired location $x_{i,s}$ of an intensity value of a specific split at landmark point i as follows:

$$x_{i,s} = \begin{bmatrix} \text{rect}_{\text{width}} & 0 \\ 0 & \text{rect}_{\text{height}} \end{bmatrix} \times (M_{\text{s}} \times \begin{pmatrix} \delta_x^{i,s} \\ \delta_y^{i,s} \end{pmatrix} + \begin{pmatrix} x_{i,x} \\ x_{i,y} \end{pmatrix}) + \begin{pmatrix} \text{rect}_x \\ \text{rect}_y \end{pmatrix} \tag{4.22}$$

Lets assume a facial landmark in the mean shape with coordinates $x_{i,x} = 0.5$ and $x_{i,y} = 0.5$, and an offset for the desired split from these coordinates of $\delta_x^{i,s} = 0.03$ and $\delta_y^{i,s} = 0.01$. Furthermore assume that we have found a rectangle of size 100×100 at position $(\text{rect}_x, \text{rect}_y) = (200,100)$ and a rotation between two consecutive shapes of $\alpha = 45°$. Then the above equation becomes

$$x_{i,s} = \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix} \times (\begin{bmatrix} \cos 45° & \sin 45° \\ -\sin 45° & \cos 45° \end{bmatrix} \times \begin{pmatrix} 0.03 \\ 0.01 \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}) + \begin{pmatrix} 200 \\ 100 \end{pmatrix} = \begin{pmatrix} 252.83 \\ 148.59 \end{pmatrix} \tag{4.23}$$

Now that all individual parts of the algorithm are described, we can finally form the first naive version of the algorithm as shown in Algorithm 4.8 . Note that the similarity transform only needs to be computed once per iteration, and that we precompute pixel values at the possible split locations and store them in `pixelValues`. During the regression testing, we store resulting leaf indices of individual decision trees in the `leafIndices` array.

Algorithm 4.8 already points out several possibilities for parallelization. In the following Sections we will describe how this algorithm is implemented on the smartphone's GPU.

4.4.2 Training & Data Set

For our data set we resorted to using the regressors trained with the popular DLib library [60], which includes a wide range of machine-learning algorithms for training similar problems. It also includes the training methodology for the our implemented algorithm of [21]. The shape predictor was trained using the iBUG 300-W face landmark dataset [4] and contains 15 consecutive cascades or stages. Each stage contains 500 individual decision trees of depth 4, i.e. resulting in $2^4 = 16$ possible leaf nodes and 15 possible splits on each decision tree. Each leaf node then contains 68 tuples of offsets, one for each facial landmark. Each decision tree is accompanied by an anchor index, i.e. indicating which facial landmark point to take as reference for splits, as well as an array of delta values of the same size. After training in DLib, we write the values to a custom binary format which can be directly read into GPU buffers without preprocessing. The resulting arrays after training are shown in Table 4.2 .

The mean shape consists of pairs of (x,y) coordinates, while the list of anchors is simply

Algorithm 4.8 Naive Facial Feature Detection Algorithm

```
procedure DETECTFEATURES
    currentShape ← initialShape
    for *iteration* in numberOfCascades do
        $M, t$ ← getSimilarityTransform(initialShape, currentShape)
        $M_{\text{rect}}, t_{\text{rect}}$ ← getRectangleTransform(faceBoundingRectangle)
        pixelValues ← []
        leafIndices ← []
        for $i$ in numberOfRandomForests do
            anchor ← anchors[$iteration$][$i$]
            delta ← deltas[$iteration$][$i$]
            point ← $M_{\text{rect}} * (M * delta + currentShape$[anchor]$) + t_{\text{rect}}$
            pixelValues[$i$] ← image.read(round(point))
        end for
        for $f$ in numberOfRandomForests do
            $j \leftarrow 0$
            while $j <$ numberOfSplits do
                split ← splits[$iteration$][$f$][$j$]
                pixelValue$_1$ ← pixelValues[split.idx1]
                pixelValue$_2$ ← pixelValues[split.idx2]
                if pixelValue$_1$ − pixelValue$_2$ > split.threshold then
                    $j \leftarrow 2j + 1$
                else
                    $j \leftarrow 2j + 2$
                end if
            end while
            leafIndices[$f$] ← [$j$ − numberOfSplits]
        end for
        for $i$ in numberOfFeaturePoints do
            for $f$ in numberOfRandomForests do
                currentShape[$i$] ← currentShape[$i$] + leafValues[$iteration$][$f$][leafIndices[$f$]]
            end for
        end for
    end for
end procedure
```

a list of indices referencing facial landmarks stored as integer values. Note that for each stage different anchors are used, thus the multiplication by the number of stages $N = 15$. Deltas consist of a pair of floating-point values from facial landmarks, and are stored again as floating-point values. The split array consists of a list of split triples with each decision tree laid flat into memory sequentially. Each split consists of three values: two integer-based indices on which feature points to compute the feature on, and a floating-point based threshold. Given our tree depth of 4, we have 15 split nodes and 16 possible leaf nodes as shown in the leaf values array.

In our shaders, we use structs indicated in Algorithm 4.11 .

Code 4.11 MSL Shape Predicition Structs

```
struct ShapePredictorCounts {
    int nFeatures;
    int nAnchors;
    int nAnchorsPerForest;
    int nDeltas;
    int nDeltasPerForest;
    int nForests;
    int nForestsPerForest;
};
struct ShapePredictorSplit {
    int idx1;
    int idx2;
    float threshold;
};
```

The first struct, `ShapePredictorCounts` is constant and does not change on individual frames and is denoted as the variable `param` in the following code examples. The second struct, `ShapePredictorSplit` is dynamically put together when required. Algorithm 4.12 shows the functions used in our MSL shaders to access individual array elements and thus show again our data layout. Note that `iter` denotes the current iteration in terms of cascades.

We use the `packed_float3` Metal data-format in order to avoid storing four unnecessary bytes, given that the native `float3` type is 16-bytes aligned [18].

4.4.3 Parallel Algorithm

Similar to Algorithm 4.8 , our parallel approach consists of four steps for each cascade:

- Calculate Similarity Transform
- Gathering of Feature Pixel Values
- Regressing
- Updating the current shape

All of these three steps contain possibilities for parallelization. Our overall approach looks as follows: in our shape predictor, we first retrieve the number of detected rectangles from

Code 4.12 MSL Shape Predicition Accessing Functions

```
int getAnchorIndex(int iter, int i, const device int*
   anchorIndices) {
    return anchorIndices[iter * param.nAnchorsPerForest + i];
}

float2 getDelta(int iter, int i, const device float2* deltas) {
    return deltas[iter * param.nDeltasPerForest + i];
}

float2 getLeafValue(int iter, int i, int leafNum, int
   featurePointNum, const device float2* leafs) {
    return leafs[iter * (param.nForestsPerForest * (param.
       nForests + 1) * param.nFeatures) + i * ((param.nForests +
       1) * param.nFeatures) + leafNum * param.nFeatures +
       featurePointNum];
}

ShapePredictorSplit getSplit(int iter, int i, int splitNum,
   const device packed_float3* splits) {
    float3 _split = splits[iter * param.nForestsPerForest *
       param.nForests + i * param.nForests + splitNum];
    ShapePredictorSplit split;
    split.threshold = _split[2];
    split.idx1 = as_type<int>(_split[0]);
    split.idx2 = as_type<int>(_split[1]);
    return split;
}
```

the haar cascade object detection process along with a buffer containing the position x and y as well as width and height of the respective rectangle. We then fill a dispatch buffer with the number of rectangles. We will use one threadgroup per found rectangle, thus use the `dispatchThreadgroups(indirectBuffer)` method of our `MTLCommandBuffer` to utilize the dispatch buffer as argument for the number of issued threadgroups. In each threadgroup, we utilize the maximum number of available threads (512 in the case of our shader on an iPhone 7) to compute the facial features within the respective rectangle. We then loop through the number of cascades and call our compute shader once for each stage, thus carrying the current shape forward on device-based memory. The first step in each iteration is the calculation of the similarity transform. The parallellized process for this computation is explained in Section 4.2.4 . We then compute the feature pixel values at the respective offsets.

Gathering of Feature Pixel Values

As required feature pixel values for the regressors can be computed individually, we divide the number of feature pixel values by the number of available threads and let each thread handle its share of feature pixel values. We make use of the accessor defined in Code 4.12 to access pixel deltas and anchor indices. The code for this stage is shown in Code 4.13 .

Code 4.13 MSL Feature Pixel Gathering

```
float2 rectTrans = float2(rect[blockIdx][0],rect[blockIdx][1]);
float2x2 rectRot = float2x2(0);
rectRot[0][0] = rect[blockIdx][2];
rectRot[1][1] = rect[blockIdx][3];

for(int i=threadIdx; i<params.nDeltasPerForest; i+=
   numberOfThreads) {
    float2 delta = getDelta(iteration, i, deltas);
    int anchor = getAnchorIndex(iteration, i, anchors);
    float2 local = similarityRot * delta+currentShape[anchor];
    float2 p = round(rectRot * local + rectTrans);

    if (p.x >= 0 && p.y >= 0 && p.x < width && p.y < height) {
        featurePixelValues[i] = round(grayscale.read(uint2(p.x,p
           .y)).x * 255.0);
    } else {
        featurePixelValues[i] = 0.0;
    }
}
threadgroup_barrier( mem_flags::mem_threadgroup );
```

Metal natively supports multiplication and additions on small matrices and vectors, so we are not required to implement those directly. We store all retrieved pixel values from our grayscale image in a threadgroup memory named `featurePixelValues`, so they can be accessed very rapidly in later stages. We also round the retrieved coordinates in `p` to the nearest pixel and check for out-of-bounds in case of deformed transformations. The last remaining step in this processing stage is to issue a `threadgroup` command with the `mem_flags::mem_threadgroup` flag in order to wait for all threads having written their values into shared memory.

Regressing

In the regression stage, we make use of the fact that each decision tree can be computed individually. Code 4.14 shows how an individual leaf node is found.

Code 4.14 MSL Random Forest Regression

```
for(int i=threadIdx; i<params.nForestsPerForest; i+=
   numberOfThreads) {
    int j = 0;
    while(j < params.nForests) {
        ShapePredictorSplit split = getSplit(iteration, i, j,
           splits);
        if (featurePixelValues[split.idx1] -
            featurePixelValues[split.idx2] > split.threshold) {
            j = 2*j + 1;
        } else {
            j = 2*j + 2;
        }
    }
    int leafNum = j - params.nForests;
    leafIndices[i] = leafNum;
}
```

Similar to Algorithm 4.8 , we traverse the respective tree until a leaf node has been found, and store the respective index of the leaf in another threadgroup-based memory called `leafIndices`. We can now safely and rapidly access the previously computed `featurePixelValues` from threadgroup memory in each split node.

Shape update

After having found each individual leaf index for all the decision trees in the current stage, the remaining thing to do is to update the current shape S_t. Each found leaf index contains 68 values which need to be added to the current shape S_t. Code 4.15 shows how this is parallelized.

We basically iterate through each facial landmark and let all threads in the threadgroup enter each iteration. Within that iteration, we then again parallelize in the number of decision trees to retrieve the leaf value for the feature with index `i`. Each thread then computes the sum of to-be-added deltas of its share of leaf values. This value is then subreduced to retrieve the final sum which needs to be added to the `currentShape`. In our shader, we process two features at once, i.e. four float values, in order to avoid having to do multiple subreductions and to save in terms of operations count. Finally, the first thread writes out the sum of deltas to the device-based variable `currentShape`, which then contains the desired facial landmarks,

Code 4.15 MSL Current Shape Update

```
int nFeaturesDiv = (params.nFeatures+1)/2;
for (int i=0; i<nFeaturesDiv; i++) {
    int idx = i * 2; float4 currentFeatures = 0.0;
    for(int j=threadIdx; j<params.nForestsPerForest; j+=
       numberOfThreads) {
        if (idx < params.nFeatures) {
            float2 xy = getLeafValue(iteration, j, leafIndices[j
               ], idx, leafs);
            currentFeatures[0] += xy.x; currentFeatures[1] += xy
               .y;
        }
        if (idx+1 < params.nFeatures) {
            float2 xy = getLeafValue(iteration, j, leafIndices[j
               ], idx+1, leafs);
            currentFeatures[2] += xy.x; currentFeatures[3] += xy
               .y;
        }
    }
    float4 featureSum = subReduceAdd(threadIdx, numberOfThreads,
        currentFeatures, sharedMemory);
    if (!threadIdx) {
        if (idx < params.nFeatures) {
            currentShape[idx] += float2(featureSum[0],featureSum
               [1]);
        }
        if (idx+1 < params.nFeatures) {
            currentShape[idx+1] += float2(featureSum[2],
               featureSum[3]);
        }
    }
    threadgroup_barrier( mem_flags::mem_device );
}
```

and at the end of each iteration we wait for the first thread to finish to writing operation until the next iteration step continues.

4.5 Optical Flow Intermediate Frame Tracking

4.5.1 Overview

Using our cascaded haar objection detection approach on the entire image in every frame of a video stream is computationally expensive. Given a successful face and facial landmark detection from a previous frame, one can safely assume that the position of the face as well as landmarks have not significantly changed when the processing frame rate is high enough. Thus rather than researching the entire image for potential positions of faces, we use an optical flow tracking algorithm in order to detect small motions in intermediary frames and then subsequently analyze those motions in order to transform the previously detected bounding rectangle of the previous frame. Figure 4.23 gives an example of this process visually.

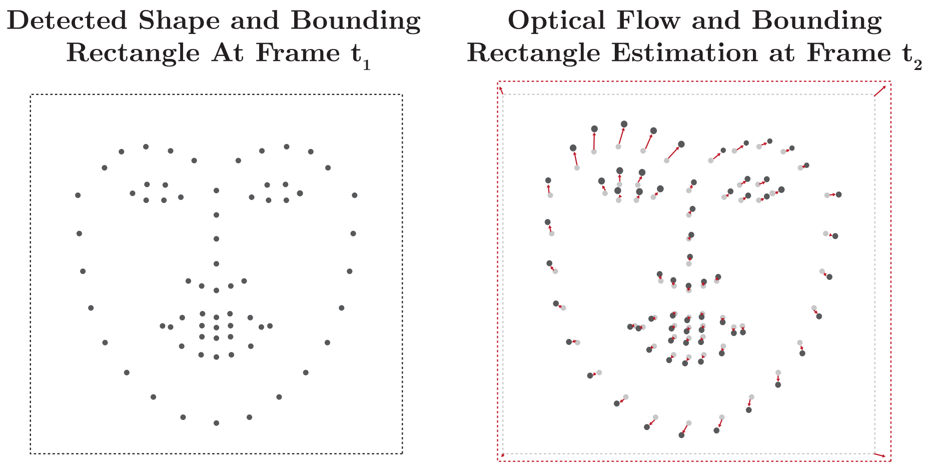

Detected Shape and Bounding Rectangle At Frame t_1
Optical Flow and Bounding Rectangle Estimation at Frame t_2

Figure 4.23: Estimation of Face Bounding Rectangles via Optical Flow

Thus the process in this thesis is as follows: we first try to detect a face rectangle R and shape S in the first frame. If detection is successful, we perform optical-flow tracking of our face landmarks for K consecutive frames. Optical-flow tracking outputs an offset vector (or sometimes referred to as velocity vector) $\vec{v} = (v_x,v_y)$ for each of the landmarks, and we use the computed similarity transform M^e, t^e between shape S and $S^e = S+\vec{v}$ to transform the gather $R^e = RM^e+t^e$ while excluding rotation from the estimated rotation M^e. This rectangle R^e is then used as initial bounding rectangle for landmark detection. After K consecutive frames, we then perform cascaded object detection again and therefore restart the process.

As stated in Section 3 , we use the Lucas-Kanade [61] optical-flow tracking algorithm to calculate the offset vector $\vec{v}$ for each of the facial landmarks. The general problem of the optical-flow algorithm can be stated as shown in Equation (4.24) [62].

$$\epsilon(\vec{v}) = \epsilon(v_x, v_y) = \sum_{x=\vec{x_x}-w_x}^{\vec{x_x}+w_x} \sum_{y=\vec{x_y}-w_y}^{\vec{x_y}+w_y} (I^t(x,y) - I^{t+1}(x+v_x,y+y_y))^2 \tag{4.24}$$

For every position $\vec{x}$ we try to find an offset vector $\vec{v}$ that minimizes $\epsilon(\vec{v})$ within a so-called integration window of dimensions w_x and w_y. The window size we have chosen for our implementation is $w_x = 10$ and $w_y = 10$ and was determined experimentally - thus a total window size of $(2w_x + 1) \times (2w_y + 1) = 21 \times 21$. The following Section describes the implementation in terms of how this problem is approached.

4.5.2 Iterative Algorithm

For our implementation we generally follow the approach of Bouguet [38]. Equation (4.24) can be reformulated as follows:

$$I(\vec{x} + \vec{v}, t + 1) \approx I(\vec{x}, t) + \vec{v} \times \nabla I(\vec{x}, t) + I_t(\vec{x}, t) \tag{4.25}$$

$I(\vec{x}, t)$ denotes a pixel intensity at position $\vec{x}$ of frame at time t, $\nabla I(\vec{x}, t)$ is the spatial gradient derivative image of I, and $I_t(\vec{x}, t)$ denotes the temporal gradient image between frame I_t and I_{t-1} [62].

To compute the terms $\nabla I(\vec{x}, t)$ and $I_t(\vec{x}, t)$ we use our integration integration window , i.e. we compute gradients of pixel-location $\vec{x}$ on the area $[\vec{x_x} - w_x, \ldots, \vec{x_x} + w_x]$ and $[\vec{x_y} - w_y, \ldots, \vec{x_y} + w_y]$. Solving this equation leads to the general optical flow equation according to [38] as

$$\vec{v} = G^{-1}\vec{b} \tag{4.26}$$

where G denotes the spatial gradient $\nabla I(\vec{x}, t)$ within the integration window

$$G = \sum_{x=\vec{x_x}-w_x}^{\vec{x_x}+w_x} \sum_{y=\vec{x_y}-w_y}^{\vec{x_y}+w_y} \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \tag{4.27}$$

and $\vec{b}$ denotes the temporal gradient $I_t(\vec{x}, t)$ or image mismatch vector within the integration window

$$\vec{b} = \sum_{x=\vec{x_x}-w_x}^{\vec{x_x}+w_x} \sum_{y=\vec{x_y}-w_y}^{\vec{x_y}+w_y} \begin{bmatrix} \delta II_x \\ \delta II_y \end{bmatrix} \tag{4.28}$$

which is two-dimensional given that we only compare two consecutive frames. I_x, and I_y denote the spatial gradient in x and y direction, while δI denotes the difference in pixel values between the current image I^t and the subsequent image I^{t+1}. This is the standard Lucas-Kanade optical flow equation, and in order to retrieve accurate results it is necessary to iterate on the scheme above multiple times (given that we are computing the solution to a least-squares estimation) [38].

Let k be the iterative index, then at each iteration k we assume a guess for for $\vec{v}^k$ based on previous iteration guesses $\vec{v}^{k-1}$, with the first guess being initialized to $\vec{v}^0 = (0,0)$. The goal then is to compute the residual pixel motion vector $\vec{\eta}^k = (\eta_x^k, \eta_y^k)$ that minimizes the error function [38]

$$\epsilon^k(\vec{\eta}^k) = \epsilon(\eta_x^k, \eta_y^k) = \sum_{x=\vec{x_x}-w_x}^{\vec{x_x}+w_x} \sum_{y=\vec{x_y}-w_y}^{\vec{x_y}+w_y} (I^t(x,y) - I_k^{t+1}(x+\eta_x^k, y+\eta_y^k))^2 \tag{4.29}$$

similar to Equation (4.24) . The solution is similar to Equation (4.26) :

$$\vec{\eta}^k = G^{-1}\vec{b}_k \tag{4.30}$$

The difference between Equation (4.24) and Equation (4.30) is the term $\vec{b}_k$, which is again the mismatch vector, but warped by the optical flow vector $\vec{v}^{k-1}$ of the previous iteration in the term δI. Therefore, δI becomes

$$\delta I_k(x,y) = I^t(x,y) - I^{t+1}(x + v_x^{k-1}, y + v_y k - 1) \tag{4.31}$$

with

$$\vec{v}^k = \vec{v}^{k-1} + \vec{\eta}^k = \sum_{k=1}^{K} \vec{\eta}^k \tag{4.32}$$

Thus the term G remains constant throughout the iteration, and only the displacement vector needs to be updated according to the value $\vec{v}^{k-1}$ of each previous iteration. Figure 4.24 shows how this is done visually.

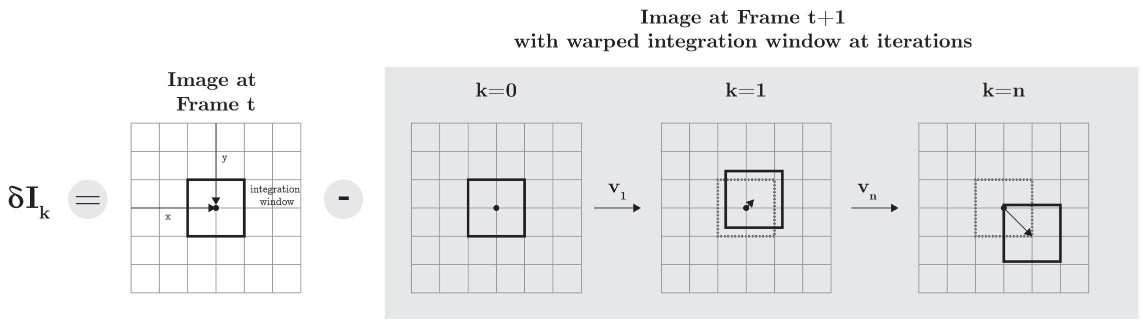


Figure 4.24: Image Mismatch Calculuation with Image Warping

While the integration window for image at frame t I^t remains constant, we warp the integration window for the subsequent frame I^{t+1} at every iteration k. On the first iteration, $v_0 = (0/0)$, we hence calculate the next value v_1 without offsetting the window at I^{t+1}. At $k = 1$, we now warp the detection window and subtract values found at the new window position from the original image I^t in order to retrieve δI. This is done until the final iteration $k = n$ is reached and the final optical-flow vector is found.

One problem that becomes apparent with this approach is that it requires sub-pixel accuracy, i.e. we need to interpolate values between pixels. We use bilinear interpolation, as being supported by the Metal platform. Figure 4.25 gives an example for this.

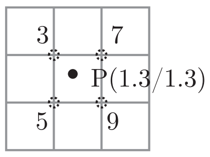


Figure 4.25: Bilinear Interpolation Example

We use the formula

$$f(p_x,p_y) = \alpha_x^i \alpha_y^i f(x,y) + \alpha_x \alpha_y^i f(x+1,y) + \alpha_x^i \alpha_y f(x,y+1) + \alpha_x \alpha_y f(x+1,y+1) \tag{4.33}$$

where $x = \text{floor}(p_x)$ and $y = \text{floor}(p_y)$ are the floored values of the original desired coordinates, $\alpha_x = p_x - x$ and $\alpha_y = p_y - y$ are the fractions between the pixels, and $\alpha_x^i = 1 - \alpha_x$ and $\alpha_y^i = 1 - \alpha_y$ are the inversed fractions. For our example from Figure 4.25 , this gives us $\alpha_x = \alpha_y = 0.3$, $\alpha_x^i = \alpha_y^i = 0.7$, $x = y = 1$ and thus

$$f(1.3,1.3) = 0.7 \times 0.7 \times 3 + 0.3 \times 0.7 \times 7 + 0.7 \times 0.3 \times 5 + 0.3 \times 0.3 \times 9 = 7.53 \tag{4.34}$$

With regards to spatial gradients, in practice, using simple forward differencing on I_x and I_y, i.e. $I_x = I(x,y) - I(x+1,y)$ will not give a consistent approximation as the x and y derivatives will be centered at different locations [63]. Thus, it is common to use the Sharr operator [64] for computing image derivatives, which can be formulated as a convolution operation.

$$I_x = \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -1 \end{bmatrix} \times I \tag{4.35}$$

$$I_y = \begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} \times I \tag{4.36}$$

The last remaining problem is the inversion of the 2×2 matrix G. In our algorithm, we follow the implementation of OpenCV [57] and check for thresholds of determinants and eigenvalues of this matrix. In case one of those values is below the threshold, we simply assume $\vec{v} = (0,0)$.

4.5.3 Pyramidal Algorithm

Choosing a small window size, such as in our implementation, has the advantage of not smoothing out details contained in the images [38]. However, the tradeoff in this case is the loss of being able to track large movements. Thus we employ a multi-scale implementation of the above mentioned algorithm by applying the above-mentioned algorithm to an image pyramid. We use a gaussian convolution kernel $c_k = ww^T$, where $w = [\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16}]$ in between down-sampling steps. A downsampling step then simply consists of removing all odd rows and columns, thus defining the next level I_{L-1} in the image pyramid with half the width and height of the original image. In our implementation, we use $L = 3$ levels, with the original image L_0 representing the original scale, and L_2 representing the smallest scale.

We use the same methodology for updating our guess as for iterations k within a single scale. We start from the smallest scale, L_2, and store the guess g^{L-1} for the next scale.

$$g^{L-1} = 2(g^L + \vec{v}^k) \tag{4.37}$$

with g^2 being initialized to (0,0). We multiply the values by two in order to account for the fact that the next image on the pyramid to be processed is twice as large as the current one, thus an estimate for an offset needs to be transformed accordingly. Therefore, in a single iteration k of scale L, we compute the image difference $\delta I_k(x,y)$ between the previous image I_L^t and the next image I_L^{t+1} on the current scale as follows [38]

$$\delta I_k(x,y) = I_L^t(x,y) - I_L^{t+1}(x + g_x^L + v_x^{k-1}, y + g_y^L + v_y^{k-1}) \tag{4.38}$$

Figure 4.26 shows this iterative process visually.

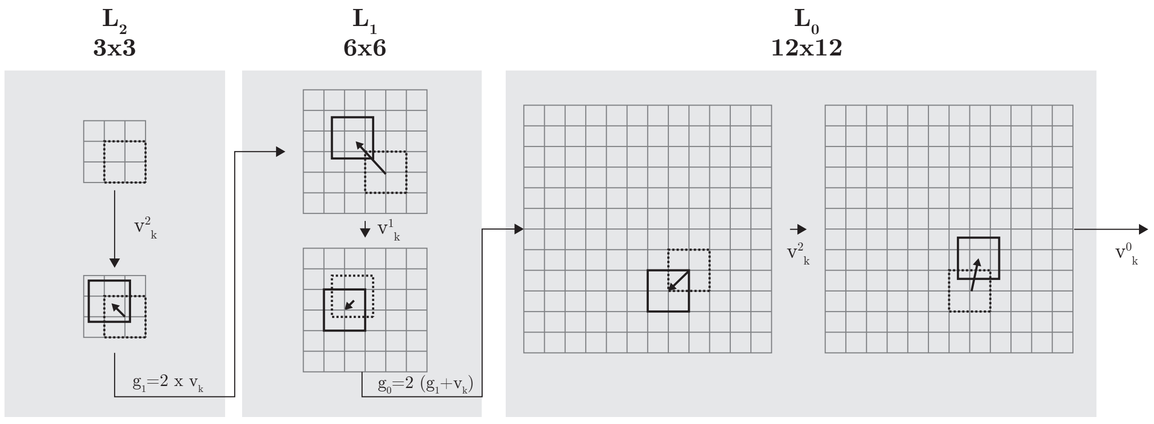


Figure 4.26: Pyramidal Optical Flow $\vec{v}$ update

After each computed scale, we pass forward the guess for the next scale $L-1$ g^{L-1}, which is then used to warp the detection window on image I_L^{t+1}. When pyramid at $L = 0$ is reached, the final optical flow is computed as

$$\vec{v} = g^0 + d^0 \tag{4.39}$$

4.5.4 Implementation

The implementation of the optical-flow algorithm ties in with the output of the facial landmark detection. The goal of this step is to receive an updated bounding rectangle of the face according to the movements of our 68 face landmarks. We will achieve this using the following steps:

1. Creation of the Gaussian Image Pyramid
2. Gathering of values for I_x, I_y and I in shared memory
3. Subreduction of values to compute matrix G
4. Check for inversibility of G
5. Iterate and update vector $\vec{v}$
6. Perform a similarity transform to update the bounding rectangle

Prior to computation of the optical flow, we need to compute the image pyramid in order to be able to process different scales. Steps 2-5 will be issued in a single shader, using a single threadgroup per facial landmark point. Thus, having 68 face landmarks, we issue 68 threadgroups using a thread block size of 16×16 for a single scale, which forms the maximum amount of threads possible for our compute shader on iOS due to its relative complexity. Thus, we for a maximum number of 3 levels, we issue our shader containing steps 2-5 3 times, once for each scale.

The final step will be to perform a similarity transform between the previous shape and the current shape of facial landmarks and then adjust the bounding rectangle used for the initial computation of the used landmarks.

Gaussian Image Pyramid

For the creation of the image pyramid, we use a shader provided by the iOS operating system named `MPSImageGaussianPyramid`[3]. This shader automatically convolves each image in the pyramid with the required gaussian kernel and stores the downscaled images in mip-map levels of the provided texture.

Buffers and Textures

We use previously stored grayscale images of the current and previous frame for the computation of the gaussian image pyramid. Next, we use only three more buffers throughout the whole optical-flow process:

- Previous Points Buffer (2×68 float values)
- Next Points Buffer (2×68 float values)
- Status Buffer (68 uchar values)

The previous points buffer basically contains the positions of the facial landmarks of image I^t. The next points buffer is uninitialized, and will be filled within the shader. It contains the predicted position of points after the successful computation of each scale. The status buffer indicates whether we found points where the spatial gradient matrix G is not inversible.

In each scale, we need to transform the point coordinates of the previous points buffer as they only refer to the scale of the original image. This operation is shown in Code 4.16 .

[3]https://developer.apple.com/reference/metalperformanceshaders/mpsimagegaussianpyramid

Code 4.16 MSL Previous Points Conversion to current Scale
```
float2 prevPt;
prevPt *= (1.0f / (1 << (currentLevel)));
```

In the smallest scale, i.e. $L = 2$, we set the variable `currentLevel` to 2, and thus receive a scaling factor of $\frac{1}{4}$ on $L = 2$, $\frac{1}{2}$ on $L = 1$, and 1 of $L = 0$. This basically transforms the original points into the coordinate frame of the current scale level.

After having scaled points in the previous points buffer, we now show how the next points buffer is retrieved as shown in Code 4.17 .

Code 4.17 MSL Next Points Conversion to current Scale
```
float2 nextPt;
if (currentLevel == param.maxLevels) {
    nextPt = prevPt;
} else {
    nextPt = nextPts[blockIdx.x];
    nextPt *= 2.f;
}
```

On the smallest scale, i.e. the first scale to be processed, we simply set the value of each point to be equal to the already scaled point from the previous points buffer. At the end of each scale-iteration, this value is then written out directly in case a valid optical-flow $\vec{v}$ was found. Thus, in the next scale, we now need to rescale the point from the previous scale in order to fit into the current coordinate system. This is equal to the multiplication by two as in Equation (4.37) . Rather than keeping track of offsets however as in the original equations, we directly add our optical-flow values to the `nextPt` values, which are read from a device-based memory.

Spatial Gradients & Spatial Gradient Matrix

After having read in the required points from the respective buffers, we now need to compute the spatial gradient matrix G for the current scale. This reduces to a simply convolve operation using the Sharr operators as described in Equations (4.35) and (4.36) . We use our own sampling function `tex2dSample` that retrieves the required four points from the passed image and computes the bilinear interpolation the the passt coordinates. Code 4.18 shows the computation the exemplary computation of the two derivatives I_x and I_y in MSL.

Given that matrix G does not change during the iterations over k, we only need to compute this once per scale per featurepoint. In our shader, we let each thread in x and y direction of our threadgroup process $\frac{2*w.x+1}{\text{blockDim}_x}$ and $\frac{2*w.y+1}{\text{blockDim}_y}$ elements in the integration window. Rather than storing the values in shared memory, we create a constant shader-based memory as shown in Code 4.19 .

The variables `dIdx_patch` and `dIdy_patch` represent the computed image derivates for each individual thread. By using the same iteration loop later again, each thread only needs

Code 4.18 MSL Sharr Operator and Image Derivative

```
float dIdx =  3.0f * tex2dSample(I, float2(x+1, y-1))
            + 10.0f * tex2dSample(I, float2(x+1, y))
            + 3.0f * tex2dSample(I, float2(x+1, y+1))
            -(3.0f * tex2dSample(I, float2(x-1, y-1))
            + 10.0f * tex2dSample(I, float2(x-1, y))
            + 3.0f * tex2dSample(I, float2(x-1, y+1)));

float dIdy =  3.0f * tex2dSample(I, float2(x-1, y+1))
            + 10.0f * tex2dSample(I, float2(x, y+1))
            + 3.0f * tex2dSample(I, float2(x+1, y+1))
            -(3.0f * tex2dSample(I, float2(x-1, y-1))
            + 10.0f * tex2dSample(I, float2(x, y-1))
            + 3.0f * tex2dSample(I, float2(x+1, y-1)));
```

Code 4.19 MSL Spatial Gradient Computation

```
float3 G(0)
float I_patch    [PATCH_Y][PATCH_X];
float dIdx_patch [PATCH_Y][PATCH_X];
float dIdy_patch [PATCH_Y][PATCH_X];

for (int yBase=threadIdx.y, i=0; yBase<=2*w.y; yBase+=blockDim.y
   , ++i) {
    for (int xBase=threadIdx.x, j=0; xBase<=2*w.x; xBase+=
       blockDim.x, ++j) {
        float x = prevPt.x + xBase - w.x, y = prevPt.y + yBase -
            w.y;
        I_patch[i][j] = tex2dSample(I, float2(x,y));
        float dIdx = ..., dIdy = ...;
        dIdx_patch[i][j] = dIdx;
        dIdy_patch[i][j] = dIdy;
        G += float3(dIdx * dIdx, dIdx * dIdy, dIdy * dIdy);
    }
}

G = subReduceAdd(threadIdx, blockDim.x * blockDim.y, G,
   sharedMemory);
```

to read from its own constant memory without the need to share its values with other threads. We also read in the actual image values at $I(x,y)$ into `I_patch` in the same fashion as they will be required for the computation of the vector $\vec{b}$ (see Equation (4.28)). The matrix G consists of four distinct values, $I_x(x,y)$, $I_y(x,y)$ and $I_x(x,y) \times I_y(x,y)$. We only store those three values and compute the sum within each thread. As G is the sum of all those values over the integration window, we `subReduceAdd` the resulting `float3` in order to retrieve the three required values to assemble matrix G.

`PATCH_Y` and `PATCH_X` are constants defined according to the size of the integration window the maximum number of threads in the used threadgroup. We check upon the valid requirements for these constants as shown in Code 4.20 .

Code 4.20 MSL Shader Memory Assertions

```
assert((2*w.x + block.width)/block.width < PATCH_X)
assert((2*w.y + block.height)/block.height < PATCH_Y)
```

Given our block dimensions of 16×16 and our window size of 10×10, this results in minimum values of `PATCH_X=PATCH_Y=3`.

Inversibility

After having computed the spatial gradient matrix G, we now need to check whether G is invertible. We use two criteria for this: the determinant of G and its eigenvalues. We use the thresholds `determinantThreshold=1e-7` and `eigenValueThreshold=1e-4` in order to check for this criteria as shown in Code 4.21 .

Code 4.21 MSL Spatial Gradient Inversibility Check

```
float D = G[0] * G[2] - G[1] * G[1];

if (D < determinantThreshold) {
    if (tid == 0 && lvl.lvl == 0)
        status[blockIdx.x] = 0;
    return;
}

float E = 0.5f * (G[0]+G[2]) - 0.5f * sqrt(4 * pow(G[1],2) + pow
   (G[0]-G[2],2) );

if (E < eigenValueThreshold) {
    if (tid == 0 && lvl.lvl == 0)
        status[blockIdx.x] = 0;
    return;
}

G *= 1.f / D;
```

If one of the conditions is not fulfilled, we return all threads for the current facial landmark and proceed to the next scale. In case we have reached the final scale, we set a flag in the `status` buffer accordingly. In order to retrieve the inverted matrix G^{-1}, we use the following standard formula for the inversion of a 2×2 matrix as shown in Equation (4.40)

$$G^{-1} = \frac{1}{\det(G)} \begin{bmatrix} \sum I_y^2 & -\sum I_x I_y \\ -\sum I_x I_y & \sum I_x^2 \end{bmatrix} \tag{4.40}$$

with the determinant being

$$\det(G) = \sum I_x^2 \sum I_y^2 - \sum (I_x I_y)^2 \tag{4.41}$$

Instead of rearranging the items of the matrix G^{-1}, we merely multiply the matrix G with the inverse determinant in the line `G *= 1.f / D` and account for the rearrangement when computing the optical flow $\vec{v}$.

Iteration and Optical-Flow Update

During the iteration process, we need to perform two operations: first, compute the vector $\vec{b}^k$ at each iteration k, and then update the `nextPt` variable storing our current guess for the location of `prevPt` in the next image I^{t+1}. In our implementation, we use `maximumIterations =30` and a `minimumDeltaThreshold=1e-2`, i.e. limit the number of limitations to 30 and stop the iterations once the optical flow vector is below the minimum delta threshold in both dimensions x and y. As an iteration k is depending on the value of `nextPt` from the previous iteration $k-1$, we cannot parallelize in terms of iterations. However, we use the same loop as in Section 4.5.4 to iterate over the integration window and assemble $\vec{b}^k$. In Code 4.22 we denote the intensity value of I^t as `I_val`, and I^{t+1} as `J_val`. The value `I_delta` then stores the image difference between the two, and we multiply that value by 32 in order to account for the fact that we use full-integer values in our Sharr operator in Section 4.5.4 , where would actually need to weigh each value by a term of $\frac{1}{32}$. It is nevertheless computationally less expensive to perform that step during the computation of the δI. Note that in each iteration k we now access the constant memory filled by each thread from the previous Section without having to recompute upon every $k+1$.

Following Code 4.22 , the next step is to compute the sums of the values each thread stored for $\vec{b}^k$. We use sub-reduction again for this task, and then add our $\vec{v}^k$ offset vector (stored as `delta`) to the `nextPt` variable, which is then passed on to the computation of larger scales. In case the `delta` value is below the specified threshold, we stop the iterations. The first thread then finally writes out our estimate of the facial feature landmark for the next scale. In case scale $L = 0$ is reached, the `nextPts` buffer contains the guesses from our optical flow algorithm for the location of the facial landmarks from frame I^t in image I^{t+1}.

Last, we use a similarity transform (see Section 4.2.4) from S_{current} to $S_{\text{estimated}}$ to translate and scale the bounding face rectangle in order to be able to compute facial features for frame I^{t+1}.

Code 4.22 MSL Single-Scale Optical-Flow Guess

```
for (int k = 0; k < maximumIterations; ++k) {
    float2 b(0);
    for (int y= threadIdx.y, i=0; y<=2*w.y; y+=blockDim.y, ++i)
       {
        for (int x=threadIdx.x, j=0; x<=2*w.x; x+=blockDim.x, ++
           j) {
            float I_val = I_patch[i][j];
            float J_val = tex2dSample(J, float2(nextPt.x + x - w
               .x, nextPt.y + y - w.y));
            float I_delta = (J_val - I_val) * 32.0f;
            b += float2(delta * dIdx_patch[i][j], delta *
               dIdy_patch[i][j])
        }
    }
    b = subReduceAdd(threadIdx, blockDim.x * blockDim.y, b,
       sharedMemory);
    float2 delta;
    delta.x = G[1] * b[1] - G[2] * b[0];
    delta.y = G[1] * b[0] - G[0] * b[1];
    nextPt.x += delta.x;
    nextPt.y += delta.y;
    if (fabs(delta.x) < minimumDeltaThreshold && fabs(delta.y) <
        minimumDeltaThreshold)
        break;
}
if (threadIdx == 0)
    nextPts[blockIdx.x] = nextPt;
```

5 Results

5.1 Goal

After having described the details of the implementation of the facial feature pipeline of this thesis, this Section aims at describing the efforts undertaken to evaluate two key metrics. First, performance as being the key differentiation from existing research and main contribution, will be the first criteria examined in the next chapter. Here, we will explain the major contributors to frame-time by describing different submodules and evaluating them individually in order to get an understanding of potential points of improvements still, as well as different settings under which the chosen algorithms work better or worse. Second, a benchmark will be outlined against a popular facial feature dataset focusing on accuracy as measured by classic metrics such as interocular distance, a measure invariant to the actual size of the images [7].

5.2 Performance

5.2.1 Overview

In order to measure performance of the chosen submodules, an iPhone 7 using an A10 Fusion chip with 64-bit architecture was used. An average of $n = 30$ tests was used to measure the frames per second (FPS) achieved in each submodule as well as the overall pipeline. The provided numbers only include raw GPU times used by a single commandbuffer and do not include CPU-time required to encode instructions which are executed on the GPU. Given a typical triple-buffer setup in production scenarios, CPU time is negligent as compared to GPU time, where most of the heavy work is done in our implementation.

5.2.2 Sub-Module Performance

Subreduction

The first separate submodule looked into in the performance evaluation is subreduction, i.e. measuring the time required to compute the sum of elements accurately and rapidly. We use the naive Swift reduce implementation as reference, e.g. `array.reduce(0,+)`. Figure 5.1 shows the results of this comparison, along with different GPU setups.

We compare the subreduction of elements ranging from 250k to 10mio elements in the graph shown above. At $n = 10^7$ elements, the naive CPU based version reaches a maximum of 0.39 FPS, while the best setup using a maximum of 64 threads and no-warp features achieves 44.78 FPS, while the second best setup using 256 threads and warp-features achieves 38.66

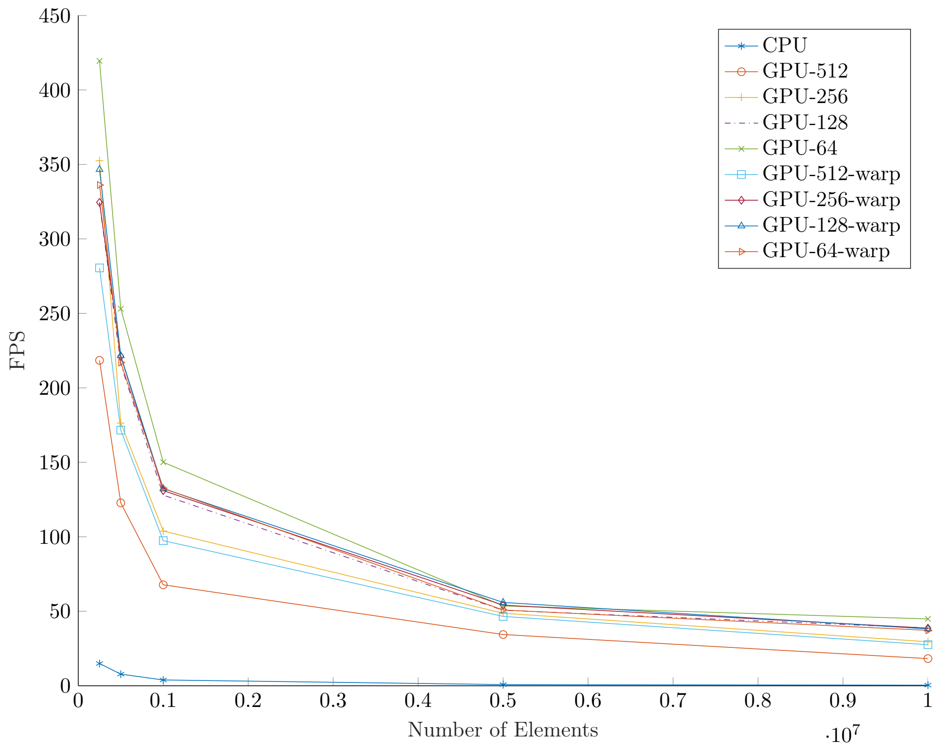


Figure 5.1: Addition-based Subreduction Comparison using CPU & GPU

FPS. It is interesting to see that the non-warp version with a much lower number of threads (and thus higher number of threadgroups) outperforms its warp-based counterpart by quite a large margin. It seems that there is a tradeoff to the fact that when using a higher number of threads per threadgroup, a large part of threads are underutilized in higher levels of the down-sweep phase of the subreduction phase. While using warp optimization to effectively eliminate the need for threadgroup-barriers can mitigate for at least stalling or blocking threads using a larger number of threads per threadgroup does only seem advantageous up to a certain threshold. This is a finetuning aspect and tradeoff that needs to be made individually for a specific problem setting.

Stream-Compaction

In this Section, we investigate the performance of our stream-compaction implementation via the same number of elements as in the previous Section. To recap, stream-compation is basically a filtering operation, e.g. reducing an array of elements to a subset given a specific criterium. For this experiment, we again utilize a naive Swift reference implementation, e.g. `array.filter { e in return e.pass > 0 }`. Figure 5.2 shows the results of this comparison employing only a single GPU-based version of $n = 64$ threads using warp-features, which outperformed other setups in our tests.

While the CPU version drops to around 0.28 FPS using $n = 10^7$ elements, the GPU-based version still manages to filter at a rate of around 16.23 FPS (around 1.7% of the GPU performance). The difference becomes even more visible in absolute terms when looking at a smaller number of elements, where the GPU based version requires far less threadblocks: when filtering 250.000 elements, the GPU version manages 293.3 FPS, while the CPU version

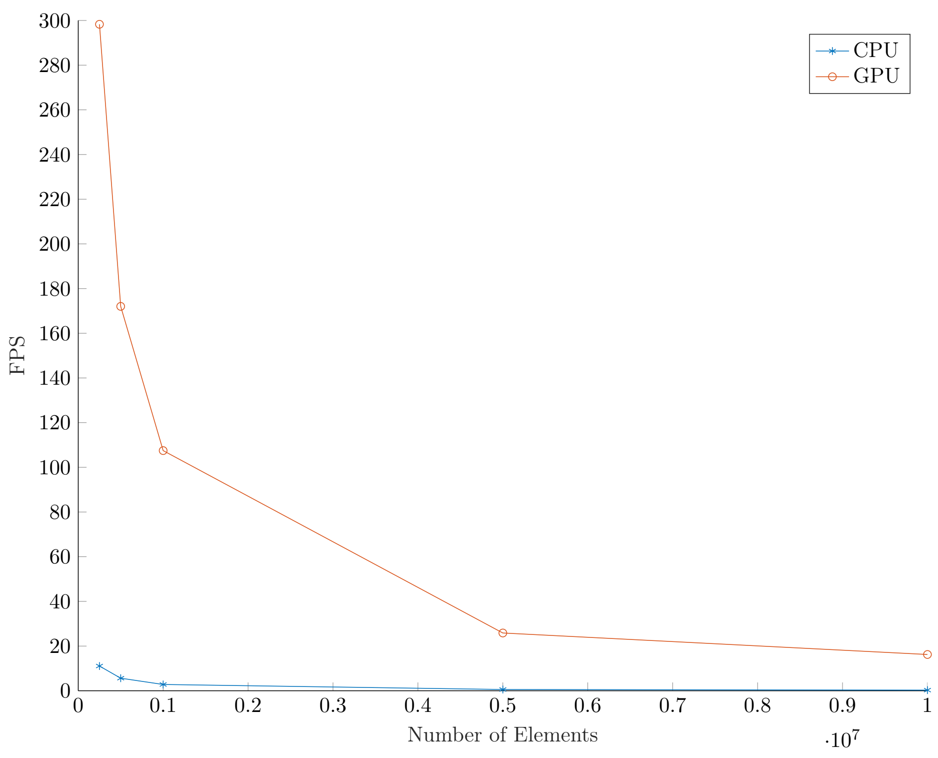


Figure 5.2: Stream-Compaction Comparison using CPU & GPU

achieves only 16.22 FPS (5.5% of the GPU performance).

Integral Image

In this Section, we estimated the performance of our integral image computation implementation on the GPU. In particular, the following typical (hi-res) phone pixel resolutions were tested: 640×480, 1280×720, 1920×1080, 3840×2160 and 4096×2160. Figure 5.3 shows the results of the implementation, with the x-axis showing the computed total number of pixels.

Even for large video input resolutions the iPhone can provide (e.g. in portrait mode) with more than 8mio pixels in total, our implementation still manages real-time processing at more than 60 FPS. As the step of computing the integral image is required once per command buffer only for the haar-cascade processing stage, this step poses no bottleneck in the overall pipeline. Given the proximity in terms of implementation as compared to the subreduction comparison, a CPU-based implementation was considered out of scope of this Chapter.

Grouping

In this Section, two highly custom modules with no reference implementations are explored that were required to bring the entire facial feature pipeline into a single GPU-commandbuffer. Figure 5.4 shows performance characteristics of the module building adjacency lists required for our rectangle grouping algorithm.

The two axes in Figure 5.4 represent again, the number of elements to group, as compared to the performance measured in FPS on the y-axis. Given different initialization parameters can lead to different difficulty in terms of finding adjacent items in a graph, four different

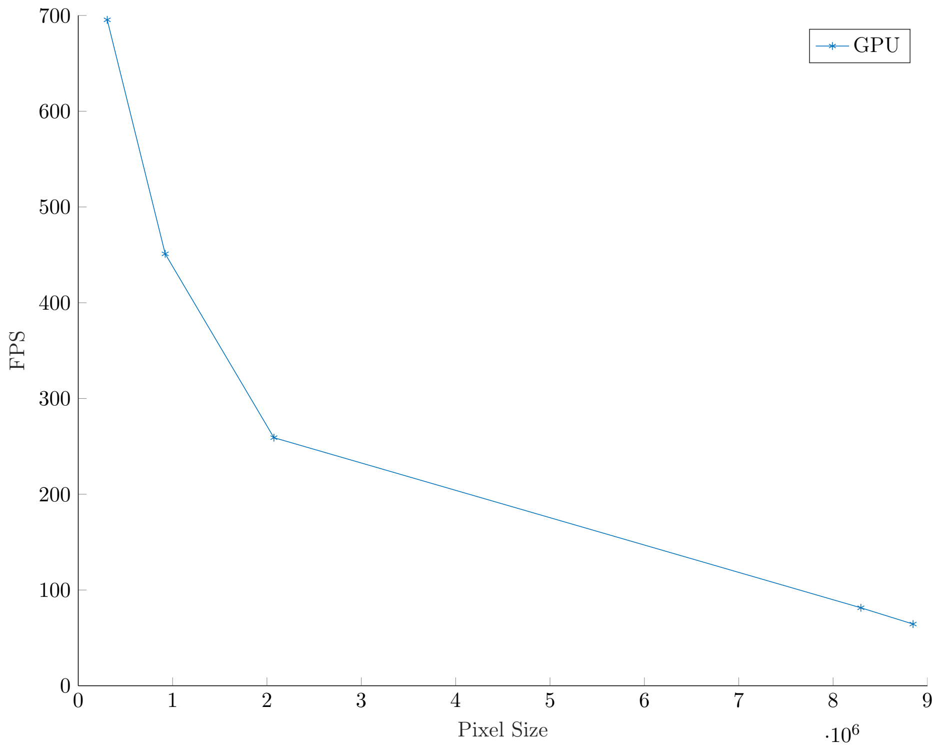


Figure 5.3: Integral Image Computation for Different Image Sizes

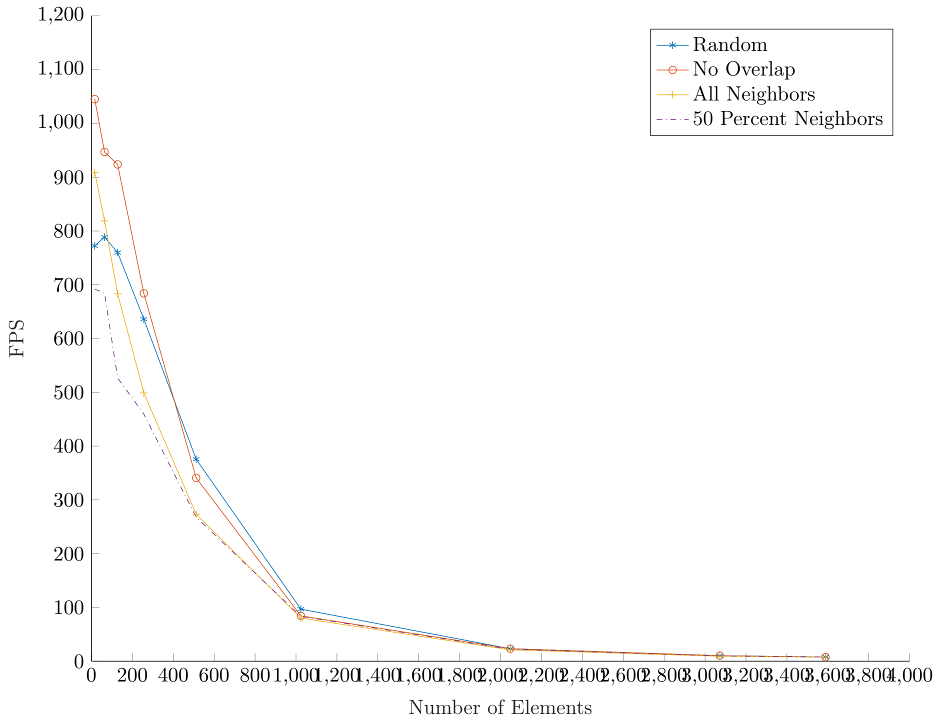


Figure 5.4: Building Adjacency Lists

setups were used to compare the performance results:

- Random - The entire dataset was created randomly
- No overlap - There is no overlap between elements, i.e. no two elements are considered neighbors

- All neighbors - All elements are considered neighbors
- 50% neighbors - Exactly 50% of elements are considered neighbors

As expected, the least computational work needs to be done - in most cases - when items have no overlap. In this case, the implementation manages around 7.8 FPS for around $n = 3500$ items. The maximum number was chosen to account for memory limitations, which apply in this scenario for the iPhone 7. Given that for rectangle grouping as implemented in the pipeline we never compare more than around $100 - 300$ rectangles, even the slowest scenarios using high or 50% overlap achieve more than 400 FPS, making the implementation sufficient for the overall real-time performance goal.

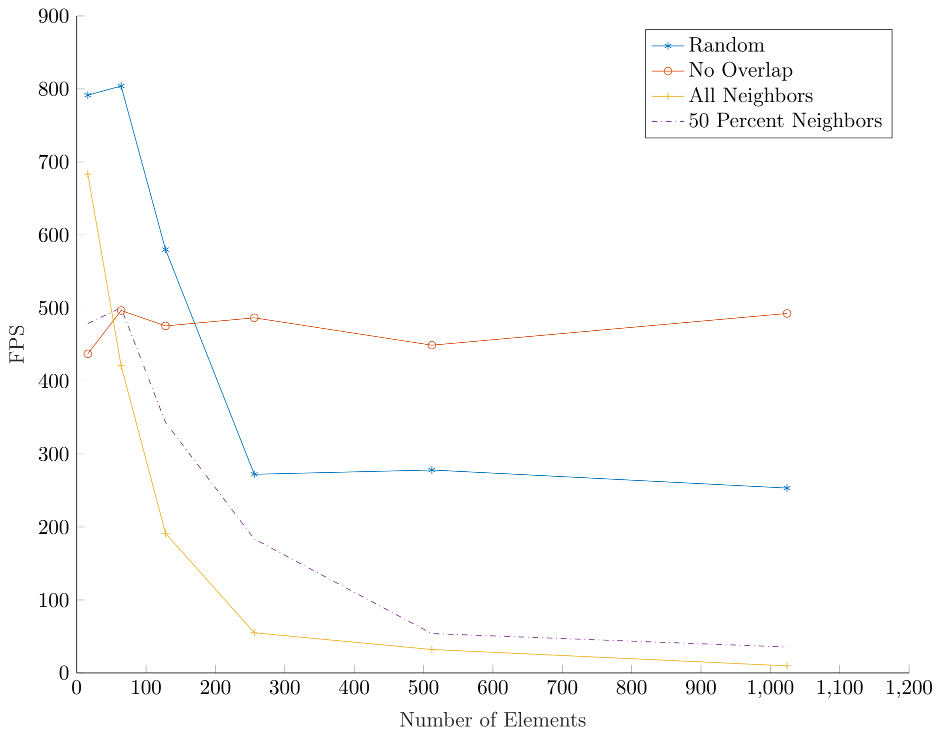


Figure 5.5: Rectangle Grouping

In Figure 5.5 , the same scenarios as in Figure 5.4 are used but for the entire rectangle grouping algorithm, e.g. from having a random set of rectangles with a specific amount of overlap, to having identified and reduced those rectangles to a subset of the original amount given a threshold. In this case, the threshold is chosen accordingly to the respective criteria in order to achieve the desired amount of overlap. It becomes clear that in case of no overlap the runtime of the algorithm becomes almost constant with increasing number of rectangles, while having placed all rectangles as neighbors in a graph-like fashion makes the implementation much slower. Grouping $n = 1024$ rectangles to a single one requires - in the worst case - almost 103 ms or around 9.9 FPS, making it unsuitable for real-time use. However, the maximum number of found rectangles is a hyper-parameter that can be adjusted for in the implementation, and when looking at randomly placed rectangles or with lower degree of overlap, the implementation still achieves around 200 FPS.

Face Detection via Haar Cascades

In this Section, the performance of the entire face detection via haar cascades is shown. This includes, to some extent, all of the submodules as described in the Chapters above, and represents the first major step in the pipeline to detect facial features and is crucial for the initialization of the facial feature detection algorithm. In this experiment, $n = 100$ tests were run using the default iPhone 7 input resolution of 720×1280 pixels. We compare two different base scenarios: one where no face is present in an image, and one where exactly one face is present. In theory, the algorithms are designed to be able to deal with multiple faces in an image as well. However, given that the entire testing was done with the goal of accurately detecting only a single face, this was considered out of scope. In addition to the the two base scenarios, step sizes of $s = 1$ and $s = 2$ were used, where in case of $s = 2$ only every second pixel is used for computation. Figure 5.6 shows the results of this evaluation, with the x-axis representing different scales of the original input resolution, which is used as input to the entire haar-cascade stage.

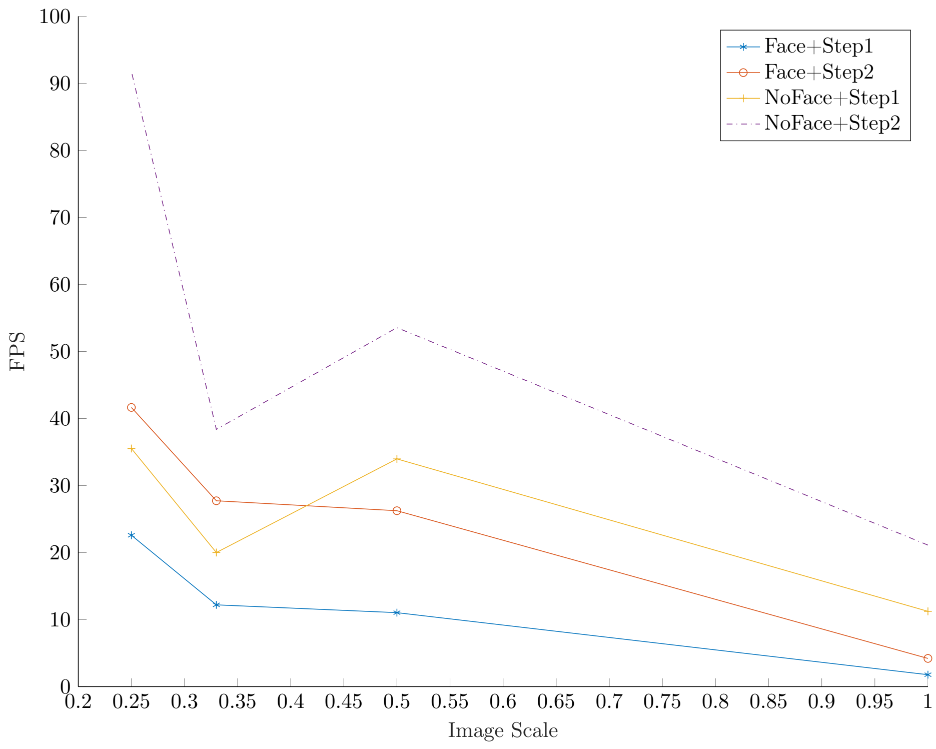


Figure 5.6: Haar Results

Overall, performance ranges from around 2FPS in the full-resolution case to 92FPS in the lowest-measure resolution. One key feature of the pipeline also becomes evident: by using and filling buffers to decide in subsequent threadgroup sizes and using them as a tool to launch different numbers of threadgroups in different stages of the haar-cascade step, the pipeline gets different runtime performances when either a face is detected or not. In case no face is detected, stages such as grouping etc. can be skipped and thus increase performance in those cases. This is particularly important in a mobile scenario, where a continuous video stream needs to be analyzed and battery can be saved in circumstances when no face is detected.

The tradeoff in terms of resolution the algorithm is working on and computation time per frame also becomes obvious when looking at Figure 5.6 : the lower the resolution the higher the resulting FPS. It is very common to use lower-resolution inputs to haar-cascade feature detection algorithms, as haar-filters only look at a sum of values over an area. Thus, depending on the input resolution, the pipeline is still able to handle all computations in real-time as will be shown in the next Chapters.

Facial Landmark Detection

After having measured and evaluated the face detection aspect of the pipeline, this Section shows the performance of the facial landmark detection implementation assuming a constant face rectangle, i.e. using no real face detection, in order to measure this stage of the pipeline separately. Similar to the previous Chapter, we use an increased number of $n = 100$ tests for this experiment, and this time alter one of the key variables relevant to the implementation: the number of iterations. Different datasets can be theoretically trained with different number of iterations to perform well for particular circumstances, e.g. cases where no facial features are to be found but other, more simple features. Figure 5.7 shows the results of this comparison.

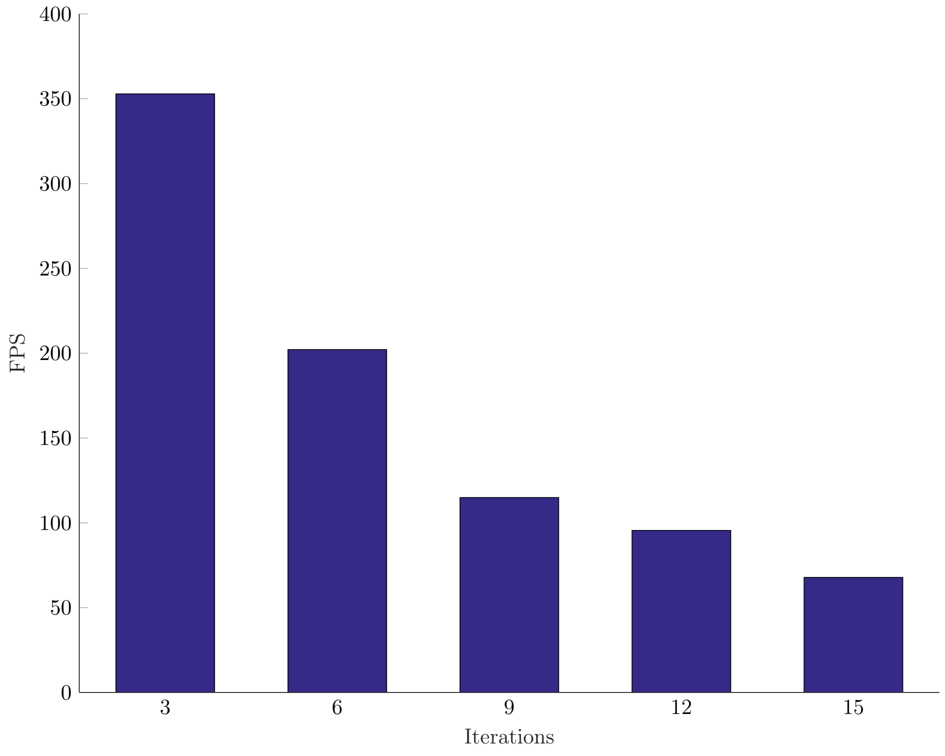


Figure 5.7: Facial Landmark Detection Performance per Iteration

For the 68-points based facial landmark detection, the implementation achieves 68 FPS for the default $i = 15$ iterations case, and 353 FPS for the $i = 3$ iterations case. Even though the smaller number of iterations is not representative in this case as it would greatly decrease the accuracy of detected facial features, it shows the non-linear increase of FPS when lowering the number of iterations. Overall, the results show that when combining our implementations of face detection with facial feature detection, we should still be able to achieve real-time performance, especially when combined with optical flow as shown in the next Chapter.

Optical Flow

Given that the face detection is computationally very intensive as shown in the previous Chapters, optical flow was introduced as described in Section 4.5 in order not to be required to run the face detection at every frame, and track the flow of the facial area and thus deform/translate the face rectangle according to the predicted optical flow. This Section evaluates the optical flow implementation on its own tracking a sparse number of points ranging from $n_1 = 100$ to $n_2 = 5000$. Three different setups are used and combined to show performance under different circumstances.

- Levels - the number of images in the image pyramid (scaling) used for computation
- Window size - the square window size used to compute the image gradients
- Iterations - the number of iterations used for solving

Figure 5.8 shows the results for the average of $n = 100$ repeated experiments.

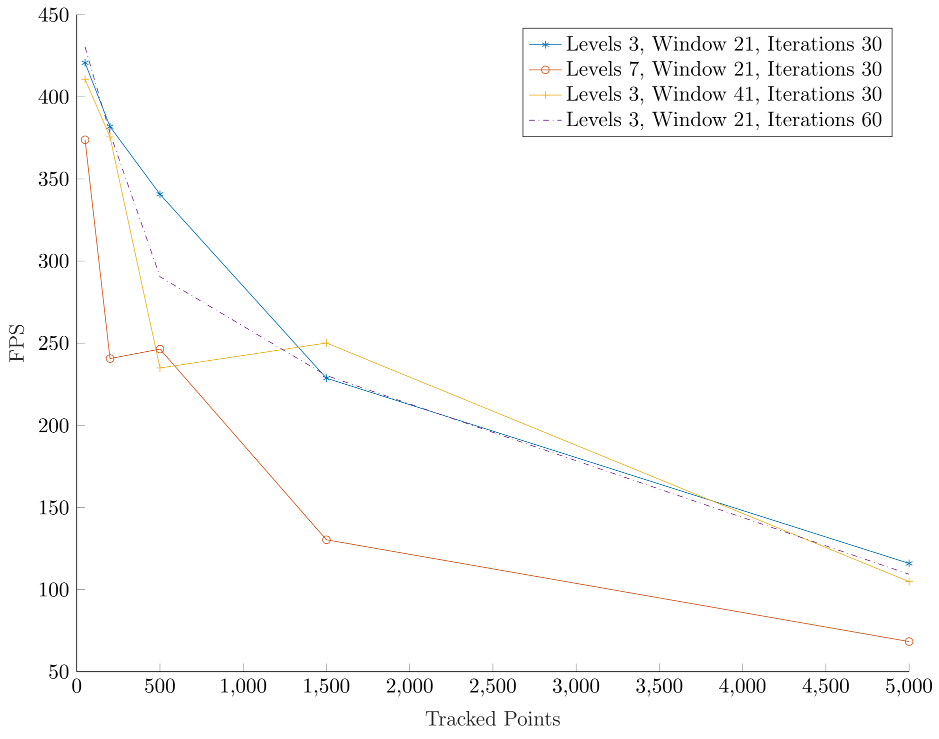


Figure 5.8: Optical Flow Performance Evaluation with increasing Number of Tracked Points

Results range from 68 to 430 FPS, with larger window size, increased number of iterations and higher number of images in the image pyramid leading to decreased performance as expected. In terms of accuracy, experiments showed no major difference between higher number of levels in the image pyramid for our use case, which naturally seems to be the most influential factor on performance. Thus, using a three-level pyramid, the optical flow implementation achieves a sufficient rate of more than 100 FPS for al other boundary cases.

5.2.3 Pipeline-Performance

In this Section, the performance of the entire pipeline, starting from input image to having found facial feature points is analyzed. In order to get an overview of the interplay between different submodules, we first ran the pipeline under three different scenarios for a time of $n = 850$ frames analyzing live-video from the iPhone 7:

- No face - no face was visible in the video for the entire time
- Face detection every frame - no use of optical flow for the entire time, face visible the entire time
- Face detection plus optical flow - optical flow was used for $t = 10$ consecutive frames, i.e. the face detection step was reinitialized every ten frames, face visible the entire time

The results of this experiment are shown in Figure 5.9 , and parameters were chosen according to the best resulting case of the submodules as described in the previous Chapters.

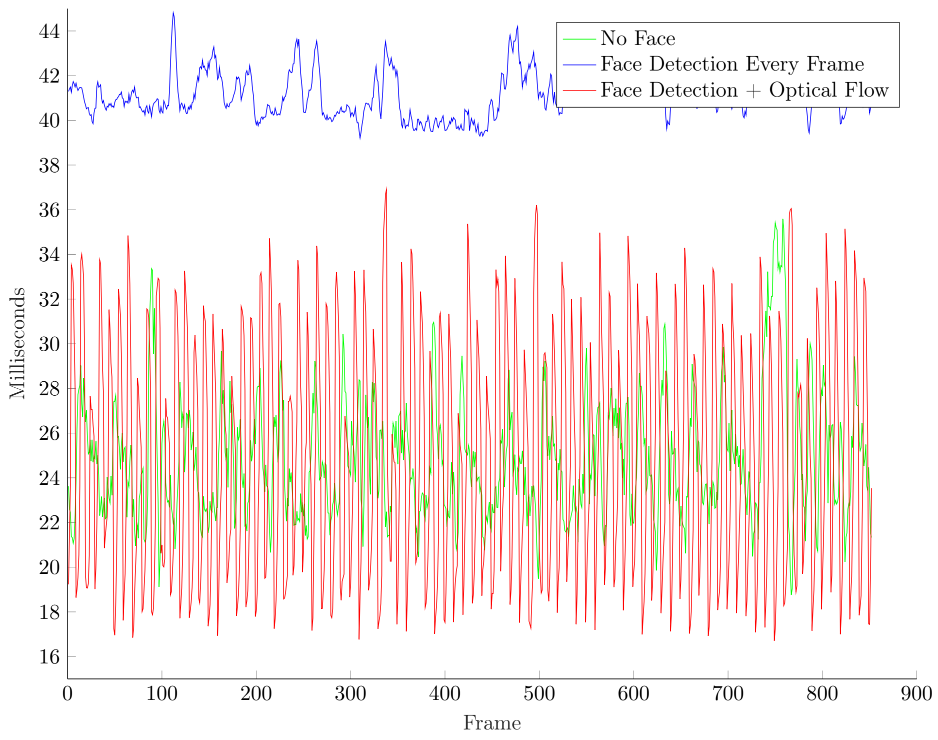


Figure 5.9: Pipeline Performance under different settings

First, using face detection on every frame (ef) results in the most computational effort on the one hand, but rather constant runtime on the other. We measured a mean of $\mu_{ef} = 41.3066$ ms or 24.20 FPS for the entire time (850) frames of the experiment. In the case of no face visible (nf), an average of $\mu_{nf} = 24.7958$ms or 40.33 FPS was measured as shown by the green line in Figure 5.9 . While the peaks for ef and nf can originate from multiple factors such as thermal throttling or image noise, the peaks in the scenario when optical flow of is used are clear: every ten frames there is a peak indicating the the face rectangle was re-detected using the haar-cascade stage. For this case, an average of $\mu_{of} = 25.0506$ms or 39,9 FPS was measured. This is a more than 60% improvement in terms of FPS as compared to the case using no optical flow, and is almost on par with the version where no face is detected and thus half of the pipeline is not triggered. Considering newer Apple devices contain even more computing power, the pipeline over-achieves its goal regarding real-time performance goals on the iPhone 7 test-device.

5.3 Validation

5.3.1 Validation Data

In this Chapter, the pipeline's accuracy will be evaluated using the popular 300 faces In-the-wild dataset (300 FITW) [4] [5] as well as the labeled faces in the wild (LFW) dataset [65] [66]. These datasets aims at covering a variety of different poses, expressions, illuminations and backgrounds and represents a good testing scenario in terms of representability for the typical phone-camera use case of our framework. The 300 FITW dataset consists of 300 indoor and 300 outdoor images respectively [4], whereas the LFW dataset contains a much larger number of face photographs from which a random subset of again 600 images was selected for testing. Images in general were collected from publicly available websites manually and then annotated by professional annotators. The resulting ground-truth for each image consists of 68 landmark points similar to well-established landmark configuration of MultiPIE [4] [67] and was already presented to the reader in Chapter 4.4 .

In general, two different aspects will be evaluated in the following Chapters:

- Face Detection - the correctness of the localization of the face in the image (LFW dataset)
- Facial Feature Detection - the correctness of locations of facial features in the image (FITW dataset)

For both experiments, a testing setup was created in which the images of the dataset were continuously pulled from a backend using the iPhone 7 test-device, and where results were then stored in a database for evaluation.

For the entire validation experiment, the same parameters were used as in the experiment described in Section 5.2.3 .

5.3.2 Face Detection

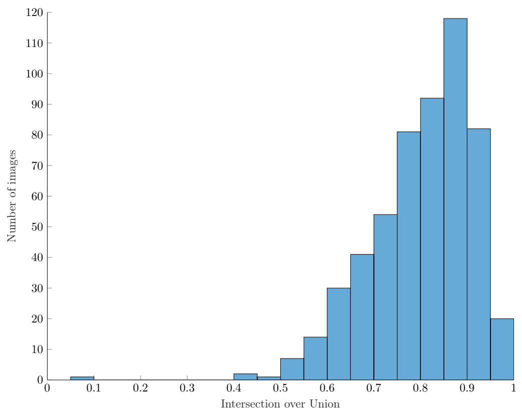
In order to compare the implementation's accuracy against the used dataset directly, the popular OpenCV [68] was used to detect faces in the dataset. In particular, the open-source stump-based 20x20 gentle adaboost frontal face detector [1] was used in both OpenCV and the implementation of the thesis: in order to create similar results, the haar-cascade parameters were converted to the internal format used by the pipeline to create comparable outcomes. Of the total of 600 images in the LFW dataset, 543 contained faces that were directly detectable by OpenCV and which were used as a basis for analysis. Images where no faces could be obtained were characterized mostly by occlusion or high rotations the face detector was not trained upon to detect.

[1] See `https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_alt.xml`

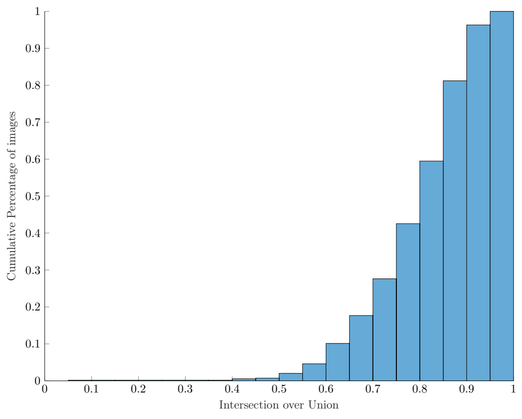
The typical metric for evaluating the accuracy of face rectangles is used, namely Intersection over Union (IOU) as defined by

$$\mathrm{IoU}(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.1}$$

Figure 5.10 shows the results of this evaluation.



(a) IOU Histogram



(b) Cumulative IOU-Comparison

Figure 5.10: Evaluation Results against the LFW [65] [66] Dataset

As can be seen in Figure 5.10a , the majority of images is centered around an IOU of 0.8 to 0.9, which is usually considered to be an excellent match - see for instance the Pascal VOC challenge [69] in the area of object detection, where best candidates achieve a (mean) IOU of 0.8.

Figure 5.10b shows the same comparison cumulatively over all images used in the computation. It becomes clear that more than half of our images are above an IOU threshold of around 0.7 to 0.8, and only a small fraction of around 0.3% has bad results in terms of IOU below 0.5.

Finally, Figure 5.11 shows detected face rectangles from our pipeline (red) and derived from OpenCV (green).

5.3.3 Facial Landmarks

In the Section the accuracy of the entire facial landmark detector pipeline will be evaluated against the 300 FITW dataset. The following measure E_i is used for each image i in the dataset:

$$E_i = \frac{\sqrt{\sum_{n=1}^{N=68} (p_{n,i}^{\mathrm{gt}} - p_{n,i}^{\mathrm{det}})^2}}{N \times (\sqrt{(p_{37,i}^{\mathrm{gt}} - p_{46,i}^{\mathrm{gt}})^2})} \tag{5.2}$$

Here, the sum of squares difference of pixel locations of all 68 facial feature points is used as the central measure of error, which is then normalized using the interocular distance [7] - the distance between eye centers as given by points with index $p_3$7 and $p_4$6 by the ground-truth points in order to be independent of the actual image size in terms of absolute error.

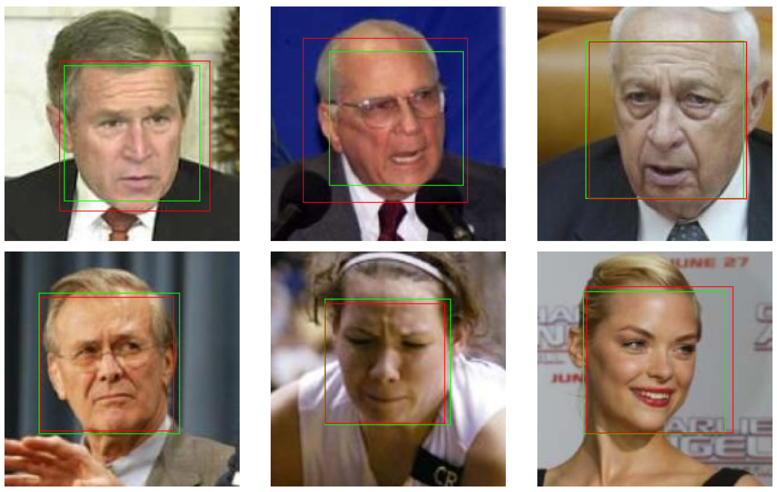


Figure 5.11: Face Detection Result Examples

Figure 5.12 shows the results of this error metric cumulatively against all images in the dataset.

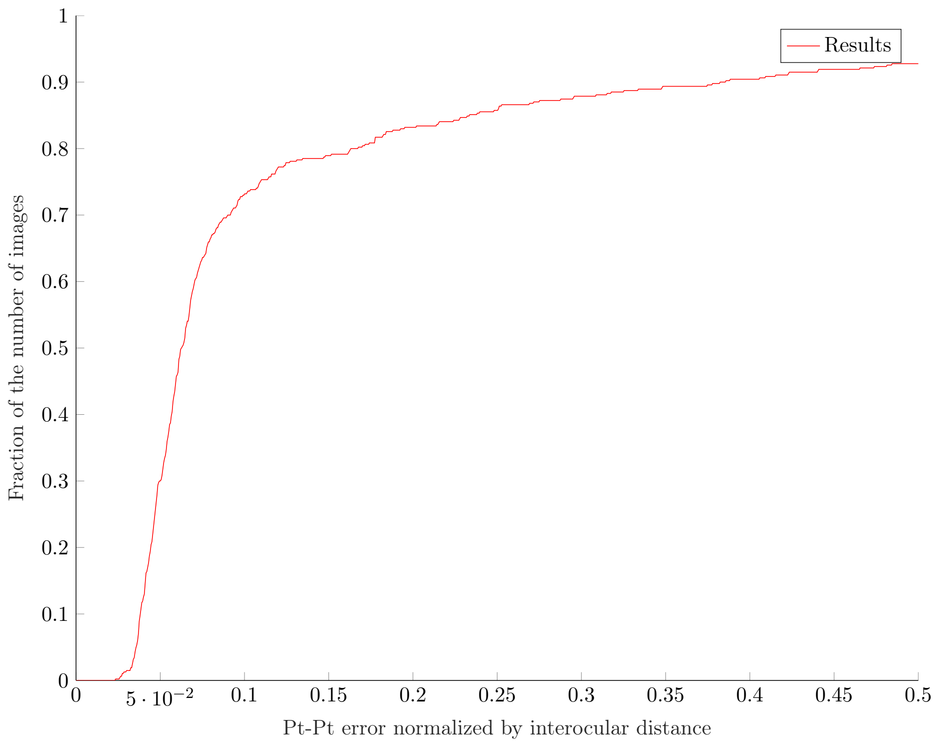


Figure 5.12: Evaluation against the 300-W [4] [5] Dataset

Overall, the pipeline managed an average error of $\mu_E = 0.1977$ over all images normalized by interocular distance, with the values ranging from the worst point difference of $E_{pi}^{w} = 6.0084$ to the closest distance of $E_{pi}^{b} = 3.2046e-4$. Around 75% of images are below the Pt-Pt error of $E = 0.1$. Compared with the best results from the 300 FITW challenge [4] [5], our work is placed in the middle-range of achieved results in the above mentioned challenge on par with the research conducted in [70], [71] and [72] accuracy -wise.

Figure 5.13 gives some impressions of a random set of images chosen from the dataset,

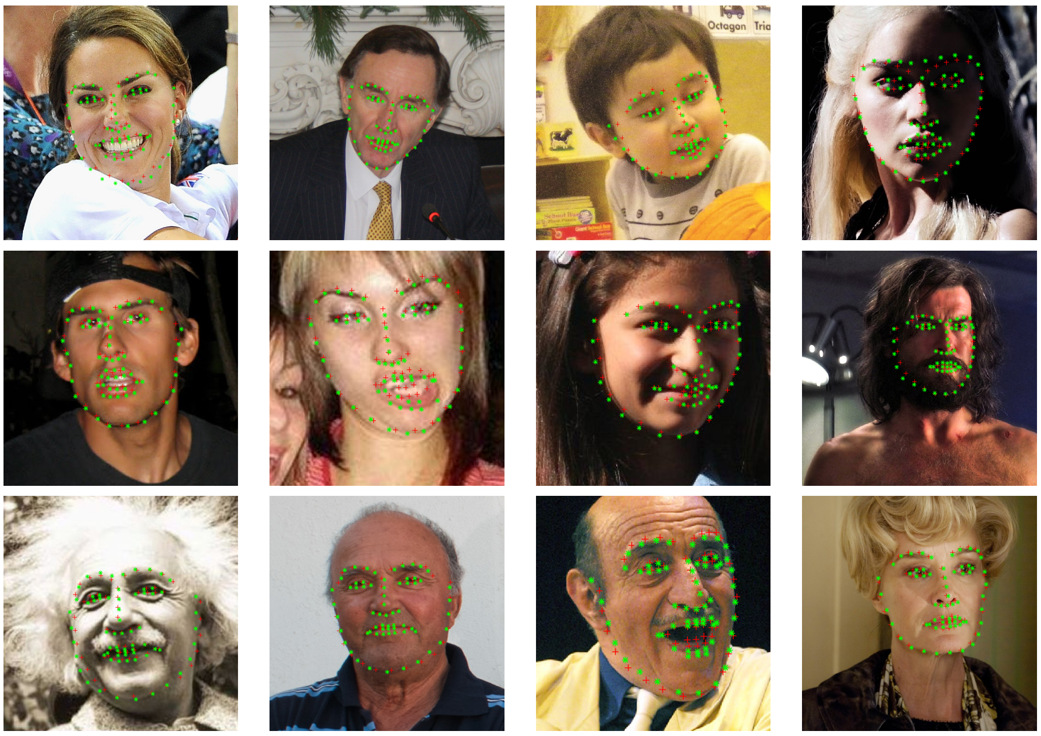


Figure 5.13: Facial Landmark Detection Results

with green dots denoting ground-truth, and red dots denoting the results of the aforementioned pipeline.

Finally, Figure 5.14 shows the six worst results in terms of the used error measure of the dataset in order to show problematic areas. Most of the large errors stem from incorrect face detection rectangles and thus incorrect initialization of the average facial feature points from which the algorithm cannot recover.

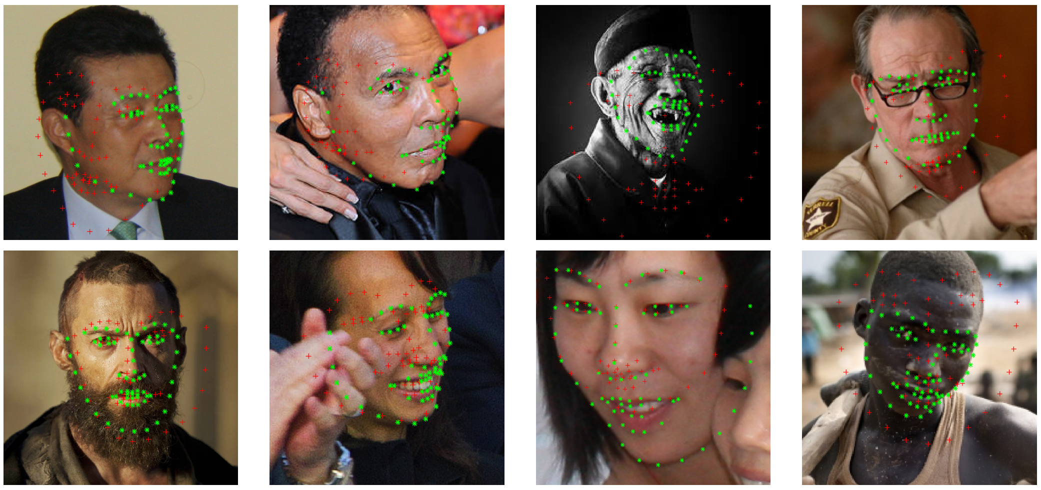


Figure 5.14: Facial Landmark Detection Worst Results

6 Conclusion

This thesis has described the approach to an implementation of a facial feature detection pipeline in images running in real-time on the iOS platform. We have designed, studied and evaluated a number of key algorithms in computer vision that were required to bring this pipeline into life from the perspective of how they potentially perform on GPUs. In additon to making a number of crucial adaptations, a series of new algorithms were provided that were essential to being able to achieve real-time performance. The facial feature detection network as the major contribution of this thesis is capable of running at around 40FPS using a 720*p* input on a modern iOS device. To put this into perspective, in our own tests using the iPhone 7 as test-device, our pipeline managed to even outperform Apple's proprietary Vision framework [1] by a fairly large margin performance-wise (using both face detection and facial feature detection at the same time, which is the recommended approach per Apple).

This can be explained by the careful selection of algorithms as well as highly-performant GPU implementations of *every* single aspect of the pipeline, requiring no computations whatsoever on the device's CPU. This selection is evidence by an extensive performance study that was conducted on the device to test every individual component for accuracy, correctness and most importantly performance in order to iteratively improve each stage of the pipeline. To the best of our knowledge, the work undertaken in this thesis is the first open-source contribution in the area specifically designed for mobile phones in the domain, and there is currently no product available achieving similar performance using only RGB images only as input.

The proposed pipeline has a number of limitations. First, given the non-constant runtime due to continuous re-detection in an image sequence with intermediate frames approximating features through optical flow could lead to spikes of high computational effort that could potentially impact user-interface computations especially on older devices. Considering the fact that work is distributed in a single command-buffer architecture in the pipeline, there is currently no way in a single GPU device to do, e.g. render computations, while the facial features are detected. A potential problem to this solution could be to split up logically separate modules with low interdependency into smaller command buffer that can be run with lower priority as compared to UI-related tasks. Second, the chosen cascaded regression forest approach to facial feature detection is highly sensitive to circumstance that did not occur during the training of said forests. In our tests we found that for example only a small number of training images contained people with glasses, making the algorithm highly sensitive and not very error prone to this particular problem. Using a larger more balanced training set to train the regression parameters and thresholds could lead to a better covering

[1] `https://developer.apple.com/documentation/vision`

of these edge cases in general. Third, even though all algorithms were implemented with maximum parallelization for the GPU in mind, certain tasks could potentially be solved more efficiently on the CPU still. Combining for instance the smaller command-buffer approach with parallel task computation could lead to lower utilization of the GPU and thus mitigate e.g. issues of thermal throttling on older devices.

Device manufacturers in particular have identified this problem and are using increasingly hardware-focused approaches to computer vision. Even though ARKit [2], along with the new iPhone X and its depth-sensing camera, still represents a tiny minority on the market, the direction of including dedicated hardware rather than software for e.g. the task of facial feature detection is a general trend that will play a crucial role in the future, especially with regards to machine-learning chips. However, the slow adoption rate of these technologies, notably in the area of budget phones, will require even more performant computer vision algorithms running on lower-quality input images for a still extended period of time, thus making it interesting to see on which area the research field in the domain will focus on in the future.

Even though classical approaches to facial feature detection have recently achieved impressive results [73], research has shown that big improvements can yet be made in better feature extraction and description [74], a process that has hitherto been approached in a try-and-error fashion. The work being done in machine learning - convolutional neural nets (CNN) in particular - seems also very promising for this area as well. Cascaded approaches have also found their way to CNNs [75], and research has already discovered ways to train entire facial feature detection pipelines outperforming state-of-the-art [76]. Fueled by the recent trend of bringing machine-learning computations to mobile phones, this could be a very interesting future development and the next big evolutional step for computer vision in general.

Considering that more and more people use phones in recent years than computers for every day tasks particularly in the younger generation [77], it remains to be seen whether a focus of research in computer vision will shift over to this new type of device comprising more and more computing power with every iteration, and whether this type of research will be able to bridge the resulting digital divide that will arise under this circumstances.

[2] `https://developer.apple.com/arkit/`

Bibliography

[1] I. Matthews and S. Baker, "Active appearance models revisited," *International journal of computer vision*, vol. 60, no. 2, pp. 135–164, 2004.

[2] Y.-Q. Wang, "An analysis of the viola-jones face detection algorithm," *Image Processing On Line*, vol. 4, pp. 128–148, 2014.

[3] W. H. Wen-Mei, *GPU computing gems emerald edition.* Elsevier, 2011.

[4] C. Sagonas, G. Tzimiropoulos, S. Zafeiriou, and M. Pantic, "300 faces in-the-wild challenge: The first facial landmark localization challenge," in *Computer Vision Workshops (ICCVW), 2013 IEEE International Conference on.* IEEE, 2013, pp. 397–403.

[5] ——, "A semi-automatic methodology for facial landmark annotation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2013, pp. 896–903.

[6] H. Zhang, Q. Li, Z. Sun, and Y. Liu, "Combining data-driven and model-driven methods for robust facial landmark detection," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 10, pp. 2409–2422, 2018.

[7] M. Dantone, J. Gall, G. Fanelli, and L. Van Gool, "Real-time facial feature detection using conditional regression forests," in *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on.* IEEE, 2012, pp. 2578–2585.

[8] F. Zhou, H. B.-L. Duh, and M. Billinghurst, "Trends in augmented reality tracking, interaction and display: A review of ten years of ismar," in *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality.* IEEE Computer Society, 2008, pp. 193–202.

[9] C. Busso, Z. Deng, S. Yildirim, M. Bulut, C. M. Lee, A. Kazemzadeh, S. Lee, U. Neumann, and S. Narayanan, "Analysis of emotion recognition using facial expressions, speech and multimodal information," in *Proceedings of the 6th international conference on Multimodal interfaces.* ACM, 2004, pp. 205–211.

[10] J. Thies, M. Zollhofer, M. Stamminger, C. Theobalt, and M. Nießner, "Face2face: Real-time face capture and reenactment of rgb videos," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2387–2395.

[11] J. Canny, "The future of human-computer interaction," 2006.

[12] NVIDIA, "Nvidia cuda c programming guide," 04 2016.

[13] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 3. IEEE, 2008, pp. 198–201.

[14] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 142–151.

[15] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. H. Wen-mei, "Cuda-lite: Reducing gpu programming complexity," in *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 2008, pp. 1–15.

[16] C. Mei, H. Jiang, and J. Jenness, "Cuda-based aes parallelization with fine-tuned gpu memory utilization," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on.* IEEE, 2010, pp. 1–7.

[17] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 152–163.

[18] A. Inc., *Metal Shading Language Specification - Version 1.2.* Apple Inc., 2016.

[19] A. Schiewe, M. Anstoots, and J. Krüger, "State of the art in mobile volume rendering on ios devices," in *Proc. EuroVis*, vol. 15, 2015.

[20] J. Sandmel, "Working with metal-overview," *Apple WWDC, published online-https://developer. apple. com/videos/wwdc/2014*, 2014.

[21] V. Kazemi and J. Sullivan, "One millisecond face alignment with an ensemble of regression trees," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1867–1874.

[22] T. F. Cootes, G. J. Edwards, and C. J. Taylor, "Active appearance models," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 23, no. 6, pp. 681–685, 2001.

[23] T. F. Cootes and C. J. Taylor, "Active shape models—'smart snakes'," in *BMVC92.* Springer, 1992, pp. 266–275.

[24] K. T. Seshadri, "Robust facial landmark localization under simultaneous real-world degradations," Ph.D. dissertation, Carnegie Mellon University, 2016.

[25] M. Valstar, B. Martinez, X. Binefa, and M. Pantic, "Facial point detection using boosted regression and graph models," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on.* IEEE, 2010, pp. 2729–2736.

[26] X. Cao, Y. Wei, F. Wen, and J. Sun, "Face alignment by explicit shape regression," *International Journal of Computer Vision*, vol. 107, no. 2, pp. 177–190, 2014.

[27] N. Dalal, "Finding people in images and videos," Ph.D. dissertation, Institut National Polytechnique de Grenoble-INPG, 2006.

[28] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1. IEEE, 2001, pp. I–I.

[29] R. Lienhart and J. Maydt, "An extended set of haar-like features for rapid object detection," in *Image Processing. 2002. Proceedings. 2002 International Conference on*, vol. 1. IEEE, 2002, pp. I–I.

[30] T. Ojala, M. Pietikäinen, and D. Harwood, "A comparative study of texture measures with classification based on featured distributions," *Pattern recognition*, vol. 29, no. 1, pp. 51–59, 1996.

[31] I. L. Kambi Beli and C. Guo, "Enhancing face identification using local binary patterns and k-nearest neighbors," *Journal of Imaging*, vol. 3, no. 3, p. 37, 2017.

[32] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 886–893.

[33] G. Dorkó and C. Schmid, "Selection of scale-invariant parts for object class recognition." in *ICCV*, vol. 1, 2003, pp. 634–640.

[34] E. Y. Chang, "Psvm: Parallelizing support vector machines on distributed computers," in *Foundations of Large-Scale Multimedia Information Management and Retrieval.* Springer, 2011, pp. 213–230.

[35] P. I. Wilson and J. Fernandez, "Facial feature detection using haar classifiers," *Journal of Computing Sciences in Colleges*, vol. 21, no. 4, pp. 127–133, 2006.

[36] H. Schneiderman and T. Kanade, "Object detection using the statistics of parts," *International Journal of Computer Vision*, vol. 56, no. 3, pp. 151–177, 2004.

[37] J. Bigun, *Vision with direction.* Springer, 2006.

[38] J.-Y. Bouguet, "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm," *Intel Corporation*, vol. 5, no. 1-10, p. 4, 2001.

[39] J. Shi and C. Tomasi, "Good features to track," Cornell University, Tech. Rep., 1993.

[40] J. Díaz, E. Ros, R. Agís, and J. L. Bernier, "Superpipelined high-performance optical-flow computation architecture," *Computer Vision and Image Understanding*, vol. 112, no. 3, pp. 262–273, 2008.

[41] B. K. Horn and B. G. Schunck, "Determining optical flow," *Artificial intelligence*, vol. 17, no. 1-3, pp. 185–203, 1981.

[42] S. D. Thota, K. S. Vemulapalli, and K. Chintalapati, "Comparison between the optical flow computational techniques."

[43] H. Nguyen, *Gpu gems 3*. Addison-Wesley Professional, 2007.

[44] G. E. Blelloch, "Prefix sums and their applications," 1990.

[45] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 407–420.

[46] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[47] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.

[48] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the conference on high performance graphics 2009*. ACM, 2009, pp. 159–166.

[49] B. Bilgic, B. K. Horn, and I. Masaki, "Efficient integral image computation on the gpu," in *Intelligent Vehicles Symposium (IV), 2010 IEEE*. IEEE, 2010, pp. 528–533.

[50] S. Umeyama, "Least-squares estimation of transformation parameters between two point patterns," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 13, no. 4, pp. 376–380, 1991.

[51] K. Baker, "Singular value decomposition tutorial," *The Ohio State University*, vol. 24, 2005.

[52] F. T. Luk, "A parallel method for computing the generalized singular value decomposition," *Journal of Parallel and Distributed Computing*, vol. 2, no. 3, pp. 250–260, 1985.

[53] J. Blinn, "Consider the lowly 2 x 2 matrix," *IEEE Computer Graphics and Applications*, vol. 16, no. 2, pp. 82–88, 1996.

[54] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical recipes in c: The art of scientific computing, cambridge," 1992.

[55] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *International Conference on High-Performance Computing*. Springer, 2007, pp. 197–208.

[56] C. H. Messom and A. L. Barczak, "Stream processing for fast and efficient rotated haar-like features using rotated integral images," *International journal of intelligent systems technologies and applications*, vol. 7, no. 1, pp. 40–57, 2009.

[57] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[58] R. Lienhart, A. Kuranov, and V. Pisarevsky, "Empirical analysis of detection cascades of boosted classifiers for rapid object detection," in *Joint Pattern Recognition Symposium.* Springer, 2003, pp. 297–304.

[59] Y. Freund, R. E. Schapire *et al.*, "Experiments with a new boosting algorithm," in *icml*, vol. 96, 1996, pp. 148–156.

[60] D. E. King, "Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, no. Jul, pp. 1755–1758, 2009.

[61] B. D. Lucas, T. Kanade *et al.*, "An iterative image registration technique with an application to stereo vision," 1981.

[62] D. Fleet and Y. Weiss, "Optical flow estimation," in *Handbook of mathematical models in computer vision.* Springer, 2006, pp. 237–257.

[63] B. Horn, *Robot vision.* MIT press, 1986.

[64] H. Scharr, "Optimale operatoren in der digitalen bildverarbeitung," Ph.D. dissertation, 2000.

[65] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller, "Labeled faces in the wild: A database for studying face recognition in unconstrained environments," University of Massachusetts, Amherst, Tech. Rep. 07-49, October 2007.

[66] G. B. H. E. Learned-Miller, "Labeled faces in the wild: Updates and new reporting procedures," University of Massachusetts, Amherst, Tech. Rep. UM-CS-2014-003, May 2014.

[67] R. Gross, I. Matthews, J. Cohn, T. Kanade, and S. Baker, "Multi-pie," *Image and Vision Computing*, vol. 28, no. 5, pp. 807–813, 2010.

[68] G. Bradski and A. Kaehler, "Opencv," *Dr. Dobb,Äôs journal of software tools*, 2000.

[69] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[70] T. Baltrusaitis, P. Robinson, and L.-P. Morency, "Constrained local neural fields for robust facial landmark detection in the wild," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2013, pp. 354–361.

[71] S. Milborrow, T. Bishop, and F. Nicolls, "Multiview active shape models with sift descriptors for the 300-w face landmark challenge," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2013, pp. 378–385.

[72] S. Jaiswal, T. Almaev, and M. Valstar, "Guided unsupervised learning of mode specific models for facial point detection in the wild," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2013, pp. 370–377.

[73] N. Wang, X. Gao, D. Tao, H. Yang, and X. Li, "Facial feature point detection: A comprehensive survey," *Neurocomputing*, vol. 275, pp. 50–65, 2018.

[74] R. Benenson, M. Omran, J. Hosang, and B. Schiele, "Ten years of pedestrian detection, what have we learned?" in *European Conference on Computer Vision.* Springer, 2014, pp. 613–627.

[75] Y. Sun, X. Wang, and X. Tang, "Deep convolutional network cascade for facial point detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013, pp. 3476–3483.

[76] Z. He, M. Kan, J. Zhang, X. Chen, and S. Shan, "A fully end-to-end cascaded cnn for facial landmark detection," in *2017 12th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2017).* IEEE, 2017, pp. 200–207.

[77] S. Zhao, J. Ramos, J. Tao, Z. Jiang, S. Li, Z. Wu, G. Pan, and A. K. Dey, "Who are the smartphone users?: Identifying user groups with apps usage behaviors," *GetMobile: Mobile Computing and Communications*, vol. 21, no. 2, pp. 31–34, 2017.