UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

Parallel Computer Vision Algorithms for Graphics Processing Units

Tesis Doctoral

Daniel Berjón Díez
Ingeniero de Telecomunicación
2016

DEPARTAMENTO DE SEÑALES,
SISTEMAS Y RADIOCOMUNICACIONES

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

Parallel Computer Vision Algorithms for Graphics Processing Units

Tesis Doctoral

Autor:

Daniel Berjón Díez
Ingeniero de Telecomunicación
Universidad Politécnica de Madrid

Directores:

Francisco Morán Burgos
Doctor Ingeniero de Telecomunicación
Profesor del Departamento de Señales,
Sistemas y Radiocomunicaciones
Universidad Politécnica de Madrid

Carlos Cuevas Rodríguez
Doctor Ingeniero de Telecomunicación
Profesor del Departamento de Señales,
Sistemas y Radiocomunicaciones
Universidad Politécnica de Madrid

TESIS DOCTORAL

**Parallel Computer Vision Algorithms
for Graphics Processing Units**

Autor: Daniel Berjón Díez

Directores: Francisco Morán Burgos y Carlos Cuevas Rodríguez

Tribunal nombrado por el Sr. Rector Magnífico de la Universidad Politécnica de Madrid, el día de de 2016.

Presidente: D.

Vocal: D.

Vocal: D.

Vocal: D.

Secretario: D.

Realizado el acto de defensa y lectura de la Tesis el día de de 2016 en

Calificación:

EL PRESIDENTE

LOS VOCALES

EL SECRETARIO

Resumen

La evolución de los teléfonos móviles inteligentes, dotados de cámaras digitales, está provocando una creciente demanda de aplicaciones cada vez más complejas que necesitan algoritmos de visión artificial en tiempo real; puesto que el tamaño de las señales de vídeo no hace sino aumentar y en cambio el rendimiento de los procesadores de un solo núcleo se ha estancado, los nuevos algoritmos que se diseñen para visión artificial han de ser paralelos para poder ejecutarse en múltiples procesadores y ser computacionalmente escalables. Una de las clases de procesadores más interesantes en la actualidad se encuentra en las tarjetas gráficas (GPU), que son dispositivos que ofrecen un alto grado de paralelismo, un excelente rendimiento numérico y una creciente versatilidad, lo que los hace interesantes para llevar a cabo computación científica.

En esta tesis se exploran dos aplicaciones de visión artificial que revisten una gran complejidad computacional y no pueden ser ejecutadas en tiempo real empleando procesadores tradicionales. En cambio, como se demuestra en esta tesis, la paralelización de las distintas subtarefas y su implementación sobre una GPU arrojan los resultados deseados de ejecución con tasas de refresco interactivas. Asimismo, se propone una técnica para la evaluación rápida de funciones de complejidad arbitraria especialmente indicada para su uso en una GPU.

En primer lugar se estudia la aplicación de técnicas de síntesis de imágenes virtuales a partir de únicamente dos cámaras lejanas y no paralelas—en contraste con la configuración habitual en TV 3D de cámaras cercanas y paralelas—con información de color y profundidad. Empleando filtros de mediana modificados para la elaboración de un mapa de profundidad virtual y proyecciones inversas, se comprueba que estas técnicas son adecuadas para una libre elección del punto de vista. Además, se demuestra que la codificación de la información de profundidad con respecto a un sistema de referencia global es sumamente perjudicial y debería ser evitada.

Por otro lado se propone un sistema de detección de objetos móviles basado en técnicas de estimación de densidad con funciones locales. Este tipo de técnicas es muy adecuada para el modelado de escenas complejas con fondos multimodales, pero ha recibido poco uso debido a su gran complejidad computacional. El sistema propuesto, implementado en tiempo real sobre una GPU, incluye propuestas para la estimación dinámica de los anchos de banda de las funciones locales, actualización selectiva del modelo de fondo, actualización de la posición de las muestras de referencia del modelo de primer plano empleando un filtro de partículas multirregión y selección automática de regiones de interés para reducir el coste computacional. Los resultados, evaluados sobre diversas bases de datos y comparados con otros algoritmos del estado del arte, demuestran la gran versatilidad y calidad de la propuesta.

Finalmente se propone un método para la aproximación de funciones arbitrarias empleando funciones continuas lineales a tramos, especialmente indicada para su implementación en una GPU mediante el uso de las unidades de filtraje de texturas, normalmente no utilizadas para cómputo numérico. La propuesta incluye un riguroso análisis matemático del error cometido en la aproximación en función del número de muestras empleadas, así como un método para la obtención de una partición cuasióptima del dominio de la función para minimizar el error.

Abstract

The evolution of smartphones, all equipped with digital cameras, is driving a growing demand for ever more complex applications that need to rely on real-time computer vision algorithms. However, video signals are only increasing in size, whereas the performance of single-core processors has somewhat stagnated in the past few years. Consequently, new computer vision algorithms will need to be parallel to run on multiple processors and be computationally scalable. One of the most promising classes of processors nowadays can be found in graphics processing units (GPU). These are devices offering a high parallelism degree, excellent numerical performance and increasing versatility, which makes them interesting to run scientific computations.

In this thesis, we explore two computer vision applications with a high computational complexity that precludes them from running in real time on traditional uniprocessors. However, we show that by parallelizing subtasks and implementing them on a GPU, both applications attain their goals of running at interactive frame rates. In addition, we propose a technique for fast evaluation of arbitrarily complex functions, specially designed for GPU implementation.

First, we explore the application of depth-image-based rendering techniques to the unusual configuration of two convergent, wide baseline cameras, in contrast to the usual configuration used in 3D TV, which are narrow baseline, parallel cameras. By using a backward mapping approach with a depth inpainting scheme based on median filters, we show that these techniques are adequate for free viewpoint video applications. In addition, we show that referring depth information to a global reference system is ill-advised and should be avoided.

Then, we propose a background subtraction system based on kernel density estimation techniques. These techniques are very adequate for modelling complex scenes featuring multimodal backgrounds, but have not been so popular due to their huge computational and memory complexity. The proposed system, implemented in real time on a GPU, features novel proposals for dynamic kernel bandwidth estimation for the background model, selective update of the background model, update of the position of reference samples of the foreground model using a multi-region particle filter, and automatic selection of regions of interest to reduce computational cost. The results, evaluated on several databases and compared to other state-of-the-art algorithms, demonstrate the high quality and versatility of our proposal.

Finally, we propose a general method for the approximation of arbitrarily complex functions using continuous piecewise linear functions, specially formulated for GPU implementation by leveraging their texture filtering units, normally unused for numerical computation. Our proposal features a rigorous mathematical analysis of the approximation error in function of the number of samples, as well as a method to obtain a suboptimal partition of the domain of the function to minimize approximation error.

Acknowledgments

First of all, I would like to thank my first advisor, Francisco Morán, for offering me the opportunity to join the Grupo de Tratamiento de Imágenes, where all the research leading to this thesis was conducted, and for his friendly guidance through several research projects and the thesis itself.

I would also like to thank my second advisor, Carlos Cuevas, who introduced me to his wonderful world of background subtraction. This thesis largely builds on his previous research, so it is no exaggeration that I could not have done it without him.

I am also grateful to Aljovsa Smolić for offering me the opportunity to do an internship at Disney Research Zürich, where I learned many things, met nice people and generally had a great time.

Of course, I also owe a debt of gratitude to all my other co-authors through these years, Rafael Pagés, Narciso García, Guillermo Gallego, Sergio García, Alexander Sorkine-Hornung, Sergio Arnaldo, Massimo Camplani, Luis Salgado and Shankar Manjunatha, all of whom have contributed with their work to this thesis and other papers.

I want to thank all my colleagues, past and present, at Grupo de Tratamiento de Imágenes because they all have made me feel so comfortably at home (maybe even too much for my own good) all these years, for all the *pausette*, football matches and interesting conversations.

I would be remiss if I forgot the nice people at Grupo de Sistemas de Tiempo Real, where I did my Master's Thesis and I began a previous PhD thesis that, even if it did not come to fruition, gave me much understanding about programming languages, computer architecture and operating systems. For all this and the nice time I spent with them I am grateful too.

I would like to thank Miguel Bordallo for all the help and friendship he has offered to me through all these years since we first started our degrees together through all the constructive talks we have had on our respective theses.

Finally, I would like to thank my family because none of this could happen without them and, most of all, I want to thank Nieves, who has endured my frustration and despair through the darkest hours of this thesis and has always offered me in return her unfaltering support and unconditional love, which have carried me through the end of this long and winding road.

Contents

1	Introduction	1
1.1	Contents	2
1.2	Contributions	3
2	The inevitability of parallelism	5
2.1	Brief history of parallel programming in CPUs	6
2.2	Graphics Processing Units	8
2.3	Heterogeneous computing, or why CPUs and GPUs complement each other . . .	11
2.4	The GPGPU computing model	14
2.4.1	Program structure	15
2.4.2	SIMT architecture and thread hierarchy	17
2.4.3	Memory hierarchy	21
2.5	Conclusion	23
3	Backward mapping DIBR for FVV	25
3.1	Introduction	25
3.2	Study of a wide baseline setup	27
3.2.1	Geometric foundation of DIBR	27
3.2.1.1	Undoing the projection	29
3.2.2	Forward mapping	30
3.2.3	Backward mapping	31
3.2.4	Filtering depth maps	33
3.2.5	Pixel colouring	33
3.2.6	Interpolation of camera parameters	35
3.3	Representation of depth data	38
3.4	Implementation notes	42
3.5	Conclusions	42
4	Real-time nonparametric background subtraction	45
4.1	Related work	47
4.2	System overview	49
4.3	Spatio-temporal nonparametric modelling	51
4.3.1	Background modeling	51
4.3.1.1	Dynamic bandwidth selection	52
4.3.1.2	Selective update	54
4.3.2	Foreground modeling	55
4.3.3	Conditioned Bayesian classifier	56
4.4	Object tracking with a particle filter	57

4.4.1	General description	58
4.4.2	Multiple object management	59
4.4.2.1	Region identification	59
4.4.2.2	Detection of new regions	60
4.4.2.3	Disappearance, split and fusion of regions	60
4.4.2.4	Redistribution of particles	61
4.4.3	Region-level tracking	61
4.4.4	Integration of the results with the nonparametric model	62
4.4.4.1	Prior probability estimation	63
4.4.4.2	Displacement of reference data	64
4.5	Selective Analysis and Regions of Interest	64
4.5.1	Detailed algorithm	65
4.6	Implementation notes	66
4.6.1	General architecture	66
4.6.2	Nonparametric modelling	67
4.6.2.1	Memory access	68
4.6.2.2	Lowering the cost of kernel evaluation	70
4.6.3	Particle filter	71
4.6.3.1	Selective analysis and regions of interest	72
4.7	Results	73
4.7.1	Parameter selection	74
4.7.2	Performance of the proposed strategy	74
4.7.3	Comparison with other detection strategies	76
4.7.3.1	SABS dataset	76
4.7.3.2	STAR dataset	78
4.7.3.3	LASIESTA dataset	79
4.8	Conclusions	79
5	Optimal piecewise linear function approximation	83
5.1	Piecewise Linear Modelization	84
5.1.1	Related Work	84
5.1.2	Basic Results in Piecewise-Linear Interpolation and Least-Squares Approximation	85
5.1.2.1	Linear Interpolation	86
5.1.2.2	Orthogonal Projection onto Vector Space V_T	86
5.2	Finding the Optimal Partition	87
5.2.1	Error in a Single Interval	88
5.2.1.1	Linear Interpolant	88
5.2.1.2	Best Linear Approximation	88
5.2.2	Approximation to the Optimal Partition	89
5.2.2.1	Procedure to Obtain a Suboptimal Partition	89
5.2.2.2	Error Estimates	90
5.3	Computational Analysis	92
5.4	Implementation in a GPU	92
5.5	Experimental Results	93
5.5.1	Gaussian Function	93
5.5.2	Lorentzian Function	96
5.5.3	Bessel Function	97
5.6	Conclusions	99

CONTENTS

6	Conclusions and future research	101
6.1	Future work	103
A	Proofs from chapter 5	105
A.1	Proof of theorem 5.1 (Linear Interpolant Error)	105
A.2	Proof of theorem 5.2 (Line Segment Minimum Error)	106
A.3	Proof of theorem 5.4 (Asymptotic Analysis)	107
	Bibliography	108

Nomenclature

ALU	Arithmetic logic unit
ANN	Artificial neural network
API	Application programming interface
CAD	Computer-aided design
CAS	Compare and swap
CIF	Common interchange format
CPU	Central processing unit
CPWL	Continuous piecewise linear
DIBR	Depth-image-based rendering
FBI	Frame buffer interface
FLOPS	Floating-point operations per second
FPU	Floating-point unit
FVV	Free viewpoint video
GMM	Gaussian mixture model
GPGPU	General-purpose computation on graphics processing units
GPU	Graphics processing unit
HDTV	High definition television
HMM	Hidden Markov model
IBR	Image-based rendering
IC	Integrated circuit
KDE	Kernel density estimation
MIMD	Multiple instruction, multiple data
OOE	Out-of-order execution

PDF	Probability density function
RISC	Reduced instruction set computing
ROP	Raster operations pipeline
SFU	Special functions unit
SIMD	Single instruction, multiple data
SIMT	Single instruction, multiple threads
SISD	Single instruction, single data
SM	Streaming multiprocessor
SPMD	Single program, multiple data
T&L	Transform and lighting
TFU	Texture filtering unit
UHDTV	Ultra high definition television
VS	Vertex shading

Chapter 1

Introduction

In the past few years, we have witnessed a profound transformation of the computing industry due to a combination of multiple factors, although arguably the catalysing event that has made the most significant impact is the emergence of ubiquitous mobile computing in the form of smart phones and tablets. These classes of devices have reached virtually everybody, including people that never before had used computers (at home or otherwise), and have finally brought the Internet to the masses, also transforming it in the process.

The incorporation of millions of laypersons to the Internet has made it an attractive channel for advertisement, forcing web sites to evolve into a much more visual media than before, which in turn has required the adaptation of access and backbone network infrastructure to be able to move much more data, video in particular. At the same time, television has switched from analogue to digital coding and transmission, blurring the boundaries between traditional television and Internet video and paving the way for a rapid quality increase: whereas analogue TV was *stuck* at the same video quality for decades, digital TV has already more than quadrupled the resolution (HDTV) of traditional TV and it is in the process of doing it again towards UHDTV (4K first and then 8K), demanding yet larger transmission channels and more computing power to efficiently treat these constantly growing data streams.

In addition to their role as terminals to retrieve contents from the Internet, smart phones are also very active contributors to it: virtually every smart phone features a digital camera, capable of capturing both photos and videos of ever increasing resolution (some current smart phones are able to capture UHDTV video). The ubiquity of smart phones has contributed to generate strong economies of scale on their components, significantly driving down the cost of digital cameras and making economical to employ them at a larger scale and/or in a wider variety of computer vision applications (surveillance, augmented reality, remote sensing, sensor networks, etc.), that can also profit from the affordable network access.

In this context, the demand for more and better video processing applications is on the rise; data sizes (resolution, frame rate) are expected to continue growing, therefore new computer vision algorithms will need to be designed to be computationally scalable. Since mobile computing is a major driving force in the industry, processors are evolving to be more energy-efficient, which has focussed hardware development on multi-core processors. Therefore, any algorithm that aims at being relevant in the medium term needs to be designed to transparently make efficient use of parallel processing architectures.

One of the most interesting such architectures at present can be found in graphics processing units (GPUs), now a standard component of every computer and virtually all smart phones, due to their formidable numeric performance and their high energy efficiency. Although these

devices were initially destined to render computer graphics, in the past few years they have been enabled for general-purpose computing (usually referred to as GPGPU); scientific computing on this kind of devices shows clear promise, but both hardware and development tools are still in a relative early stage. Therefore, in a way, today's applications may contribute to influence future development of GPUs.

In this thesis, we explore two compute-intensive application domains, such as are free viewpoint video and background subtraction, making use of the GPU to achieve real-time performance, and propose a general technique for fast evaluation of continuous functions and an efficient implementation thereof for the GPU.

1.1 Contents

In addition to the present introduction, this thesis comprises another five chapters and an appendix whose contents are briefly outlined next.

In chapter 2, the history and current status of development of general purpose processors and graphics processors are sketched. Then, the main conceptual differences between these two classes of devices are described. Finally, we present in further detail the organization of GPGPU programs, the inner architecture of modern GPUs, as well as their characteristics and limitations that need to be considered when writing code or planning memory accesses.

In chapter 3, we explore the challenges posed by wide baseline camera configurations for depth-image-based rendering, which should provide greater freedom for choosing the virtual viewpoint in a free viewpoint video context, compared with the usual camera configurations intended for use in 3D TV settings. We implement a backward mapping approach with a depth inpainting scheme based on median filters. Whilst the results back our initial assumption that this camera configuration provides good mobility, we show that the usual encoding for depth information referred to a global reference system is ill-advised and reference systems local to each camera should be used instead.

In chapter 4, we propose a background subtraction system based on nonparametric kernel density estimation methods. In this strategy, both the foreground and the background are modelled and two methods for automatic kernel bandwidth estimation, both designed to have low computational cost, are proposed. In order to offer the user easy control of the inherent tradeoff between adaptation rate and permanence of stopped objects, a selective update scheme for the background model is proposed. In addition, in order to enhance the compactness of the detection while, at the same time, reducing computational cost, a low-level tracking system based on a particle filter is added to the foreground model and a selective analysis strategy especially suited for GPU implementation is proposed. Finally, the real-time implementation of the proposed system on a GPU is detailed and extensive comparisons against other state-of-the-art alternatives are carried out over several databases.

In chapter 5, we propose a method, based on continuous piecewise linear functions, to approximate continuous functions of arbitrary complexity at constant cost on a GPU. In addition to the computational analysis and measurements, we present a rigorous analysis of the L_2 error incurred for a linear interpolator and for the best possible L_2 approximation for a given partition of the function domain. In addition, we propose a method for finding a suboptimal partition of the domain of the approximating function for minimizing distance to the original function. Finally, the proposed technique is demonstrated on several functions relevant in scientific computing. In order to facilitate the reading of this chapter, some of the proofs have been moved to appendix A. Finally, chapter 6 concludes this thesis.

1.2 Contributions

The contents of this thesis have been partially published in several peer-reviewed conferences and journals as detailed:

- A summarized version of the feasibility study on wide baseline camera configurations for DIBR view interpolation of chapter 3 can be found in [17].
- A first GPU implementation of the basic foreground and background models (fixed kernel bandwidths, blind model update) described in chapter 4 was published in [13, 42].
- A low-complexity scheme for automatic kernel bandwidth selection in the background model, a selective model update mechanism and a reduced-latency multi-threaded GPU implementation were presented in [43, 14].
- An early version of the selective analysis algorithm proposed in chapter 4, as well as the Bayesian classifier that allows combining foreground and background models with different spatial densities, was published in [15].
- A hint at future development of the background subtraction algorithm presented in chapter 4 aimed at enhancing the detection of moving objects that stop temporarily without compromising the general adaptability of the model was presented in [104].
- The technique for approximating complex functions on a GPU using its texture filtering units to evaluate continuous piecewise linear functions that is proposed in chapter 5 has been published in [16], along with the detailed error analysis for the L_2 norm; an extension of this work for the L_1 norm has also been published in [61].

Chapter 2

The inevitability of parallelism

The history of computing has been a continuous (and ongoing) quest for shorter processing times (or, alternatively, higher throughputs)¹ in all application domains. Even though the processing power of computers has steadily increased since their inception, people have always been able to think of computing problems too big or too complex to be solved within reasonable time on any of the available computers of their time. The obvious answer for this problem has always been the same: devote more units of the available resources to take on bigger tasks.

Some researchers suggested in the past that parallel computing would have limited interest. Specifically, the widely known Amdahl's law [4] states that the time $T(n)$ an algorithm takes to finish when being executed on n concurrent threads of execution corresponds to

$$T(n) = T(1) \left(B + \frac{1}{n}(1 - B) \right), \quad (2.1)$$

where $B \in [0, 1]$ is the fraction of the algorithm that is strictly sequential. This in turn means that the maximum speed-up that can be realised using several processors is bounded by said fraction:

$$\lim_{n \rightarrow \infty} \frac{T(1)}{T(n)} = \frac{1}{B}. \quad (2.2)$$

While these definitions are mathematically impeccable, they only apply if the size of the data the algorithm is applied onto is fixed; they fail to account for the fact that if more computing power is available, it might be desirable to tackle a bigger problem rather than settle for a fixed-size problem. Examples of this trend are ubiquitous: ever-growing image and video resolution, higher quality 3D models, finer-grained finite element simulations of all kinds of physical phenomena, etc. In practice, what is fixed is the execution time that is deemed reasonable for a particular application and the size of the problem is set as large as possible with the available computing resources. In particular, if the amount of parallelisable work to be done is set proportional to the number of available processors², Gustafson's law [67] applies instead:

$$\frac{T(1)}{T(n)} = n - B \cdot (n - 1). \quad (2.3)$$

¹Processing time can be defined as how long an algorithm takes to run on a given input; throughput can be defined as how many data elements can be processed per time unit.

²Note amount of work, not data size, because many algorithms have superlinear complexity with respect to the data size. In these cases, adding more processors will result in sublinear speed-up, but speed-up nonetheless.

Figure 2.1: Growth in processor performance since the late 1970s, showing a clear slowdown after 2003 due to the saturation of maximum operating frequency. © Elsevier, Inc. [75]

Thus, parallelisation may not always provide shorter execution times, but it is a practical way to tackle bigger problems. Actually, as covered in the next section, the limits of current semiconductor technology seem to hint that it well might be the only way in the foreseeable future.

2.1 Brief history of parallel programming in CPUs

Even before the invention of the microprocessor in 1971, some computer manufacturers were already building systems with several central processing units (CPUs) [106], although software tools were not developed enough to even make sequential programming easy, let alone programming of multiprocessor systems. Multiprocessor systems, however, have typically been very expensive until recently, so another solution that emerged from in-house developments [130], even before any commercial vendor offered it, was using a *cluster* of several computers connected through a network or even through primitive mechanisms such as a common disk to divide up the task(s) amongst them. Programming these clusters was extremely complex and it would take until the mid 1980s for development tools to reach the point for the first successful commercial clusters to be sold [90] and still a few more years until the development of standard networking technologies made possible open tools such as Parallel Virtual Machine (PVM) and later Message Passing Interface (MPI), which brought about the era of inexpensive and cost-effective clusters using commodity PCs.

For most users outside big enterprises or datacenters, though, all these technologies were too complex or costly and uniprocessor computers have been the norm for over thirty years. Fortunately, processor performance steadily increased every year [75] (see figure 2.1) and more and

more problems were becoming tractable over time just because newer processors were becoming so much powerful than older ones, so that single-threaded code ran faster on newer processors with no modification. However, and despite the fact that Moore's law³ has remained current through the last half century [89], the performance of single processors has somewhat stagnated in recent years because of three key factors: limited maximum power dissipation⁴, increasing divergence between memory and processing speed, and exhaustion of practical and efficient⁵ ways to exploit instruction-level parallelism (ILP)⁶. The power consideration is especially relevant given the rapid growth that mobile computing has experienced in the last decade, first with the popularisation of laptops, later tablets and smartphones.

Consequently, most general-purpose processor manufacturers have resorted in the last decade to pack more than one processing unit or *core* in each physical die⁷, so that nowadays processors with eight cores are not uncommon. Frequently, the availability of many cores is trivially exploited by the fact that in a modern multi-tasking operating system there are many processes running concurrently. However, whereas in the past newer processors ran existing code faster than older ones without any effort from the programmer, harnessing several cores for a single task requires programmers to write programs specifically designed to perform different sections of code concurrently. However, rarely if ever a task can be cleanly split into independent (and equivalent in terms of required effort) parts; much more often, different execution threads must communicate, aggregate data and wait for one another in order to yield correct results. Hence, programming (and debugging!⁸) for modern processors is necessarily more complex and error-prone, which calls for better support from development tools and higher-level constructs in programming languages to express concurrency.

For many years, most mainstream programming languages did not even bother to support concurrency at the language level, leaving to the operating systems to fill that void. Most of these, in turn, provided threads and very thin wrappers over the atomic compare-and-swap (CAS) instructions provided by the hardware in the form of mutexes, semaphores or condition variables, which made properly designing concurrent programs quite hard. It is only in recent years that mainstream programming languages have added support for higher-level constructs such as promises/futures [60, 8] (C++ 11, Python or ECMAScript 7) or monitors [76, 71] (Ada 95, Java), and parallel programming models such as fork-join [39] (OpenMP) or work stealing [19] (Intel's TBB, Apple's GCD). Undoubtedly, parallel programming will receive even more attention by the industry in years to come as the number of cores in processors continues to grow.

³Gordon E. Moore, co-founder of Intel Corp., observed in 1965 [115] that the number of components per integrated circuit (IC) that minimised per-component cost doubled every year, though he later revised this period to two years.

⁴Power dissipated by a CPU is $P = CV^2f$, where C is capacitance, f is frequency, and V is voltage. The main avenue for computing performance improvement until the early 2000s was to increase frequency, but there are physical limits to how much the other terms can be driven down with the existing silicon-based technology.

⁵Pollack's Rule (after Fred Pollack, former Intel fellow) states that microprocessor performance increase due to microarchitecture advances is roughly proportional to the square root of the increase in complexity (number of transistors) [22].

⁶Intel, the world's largest semiconductor manufacturer cancelled in 2004 ongoing projects for the two then-upcoming generations of single-core processors to focus entirely on multi-core processors [57].

⁷Unlike traditional discrete multi-processor systems, modern multi-core processors usually share some level of cache, which allows much faster inter-core communication and reduces the need to access off-chip memory, which has much lower bandwidth and higher latency.

⁸Parallel programs are notably harder to debug than sequential ones; standard practices such as breakpoints and step-by-step execution are not well defined in concurrent programs, especially for high numbers of execution threads. Some bugs are specific to concurrent programs, notably those caused by race conditions, which arise when, due to poor design, the result of a computation is nondeterministic and depends on the relative timing between interfering threads. These bugs are particularly hard to diagnose and debug because any modification to the code or execution environment (e.g., attaching a debugger) may alter the outcome, leading to their being referred to as *Heisenbugs*.

Figure 2.2: Stages of a fixed-function graphics pipeline. © Elsevier, Inc. [88]

2.2 Graphics Processing Units

As computers have been taking on more and more tasks through the years, peripheral devices have been designed for them to augment their capabilities, and the most successful of them have become staples of almost every computer, with varying degrees of integration. A commonplace feature as floating-point capabilities, for instance, was proposed a century ago [159] and first implemented in hardware in the 1940s' Zuse Z3 electromechanical computer [136] but remained somewhat of a rarity until spreadsheet applications drove the demand for faster numerical performance in the 1980s. At first, add-on floating-point units (FPUs) were sold as separate chips (Intel 8087, 80x87, Cyrix FasMath or Weitek x167 among others) that had to be explicitly programmed for. Later on, as the market found more and more applications for FPUs, they became integrated in the CPU and their instruction set assimilated as standard; today, only low-power and low-cost processors lack floating-point capabilities.

Graphics cards have followed a similar path. Before the 1980s, computers lacked a graphical interface at all and they were generally controlled through separate pieces of equipment called computer terminals, which interpreted and rendered text from a serial line or network connection. This functionality would later be integrated into expansion cards with the invention of the microcomputer. With the advent of graphical user interfaces (GUI) in the early to mid 1980s (Xerox Star, Macintosh, X Window System, etc.), graphics cards became progressively more important as users demanded higher resolutions, higher refresh rates and more faithful rendition of colours, but they would generally (outside very high-end specialised systems such as those manufactured by Silicon Graphics) remain little more than (ever growing) memory framebuffers and digital-to-analog circuitry through the mid 1990s, with the CPU of the host computer performing most of the work.

During the 1990s, the market of video games began to drive a strong demand⁹ for real-

⁹From 1985 to 2010, the video game industry grew at an annual rate of 9%–15%; global revenues in 2012 were estimated at \$67 bn [101] (cf. cinema industry at \$89 bn for the same year [1]).

Figure 2.3: Graphics pipeline with separate vertex and fragment processors. © Elsevier, Inc. [88]

time computer graphics. Many hardware manufacturers rushed to fill this new market with graphics cards that allowed the CPU to offload some of the stages of the computer graphics pipeline; many vendor-specific application programming interfaces (APIs) came along with this wave of new hardware (S3's S3d, Matrox Simple Interface, Creative Graphics Library, ATI's C Interface, PowerVR's SGL, NVIDIA's NVLIB, Rendition's Redline or 3dfx's Glide) in the hopes of establishing dominance over other manufacturers by getting developers to use their own API. By the end of the 1990s, however, Microsoft's Direct3D and Silicon Graphics' OpenGL dominated the industry and standardised both the access to hardware, freeing developers from targeting specific hardware, and the various stages of a graphics pipeline (pictured in figure 2.2).

Initially, the obvious stages of the pipeline to be moved into the graphics card were those whose basic work unit is the pixel—rastering, shading/texturing and image post-processing, which are both memory- and processor-intensive and benefit from storing the textures in a dedicated memory close to the graphic processor—, with the CPU handling all the per-vertex operations—vertex shading (VS) or transform and lighting (T&L)—. However, as the demand for more detailed 3D models grew, the CPU became more and more of a bottleneck and by 1999 the whole pipeline was implemented in hardware¹⁰ with the launch of the NVIDIA GeForce 256, which would be marketed as the world's first Graphics Processing Unit (GPU).

By this time, although Direct3D and OpenGL had mostly standardised access to hardware, the market was already demanding better visual effects and more realistic rendering. Consequently, many hardware vendors began to implement *extensions* to the standard API, mapped to their own new hardware features. The proliferation of these extensions led at first to market fragmentation because developers had to target a specific vendor if they wanted to use a specific extension, but triggered a much more interesting development: implementing such large set of features with fixed-function circuitry became uneconomical; consequently, hardware vendors

¹⁰Actually, the first hardware implementation of the pipeline predates this one by almost two decades [37]; however, this functionality would be confined to very high-end CAD systems until that moment.

Figure 2.4: Unified programmable processor array of the NVIDIA GeForce 8800 GT graphics pipeline. © Elsevier, Inc. [88]

changed their designs into programmable processors¹¹.

In 2001, both OpenGL and Direct3D were extended to allow developers to write their own programs for these new programmable stages, albeit using a low-level assembly language (OpenGL ARB assembly language/DirectX Shader Assembly Language), and in the next years new high-level programming languages were added (OpenGL Shading Language, DirectX High Level Shading Language or NVIDIA Cg, that allowed developers to write a single shader program to target both OpenGL and Direct3D), which boosted adoption of programmable shaders by the developers.

Although both geometry and fragment¹² shading was programmable, each stage was still performed by different processors, as depicted in figure 2.3. However, depending on the complexity of the scene, the screen resolution and the rendering algorithm utilised for each shader, the computing load of each stage could be very dissimilar, leading to poor utilisation of silicon resources.

In 2005, NVIDIA introduced the first unified processor for the XBox 360 console and the next year for computers (see figure 2.4 for the general layout of this device). In this design, which has become the norm since, a single set of processors with a general instruction set exists and the graphics pipeline visits it several times, so that every programmable stage in the pipeline spreads its load across the only available pool of computing resources. This change, together with the ability to select the addresses of the operands of the shaders (previously each computing thread mapped to a predefined location in a framebuffer) paved the way to use GPUs as general purpose stream processors.

The usefulness of this kind of computing device has proven such that today, as it happened before with FPU, many CPUs integrate a GPU in the same die. Actually, not only their raw performance but also their high energy efficiency have led to their integration in Systems-on-Chip (SoC) at the heart of mobile phones and other embedded computers where low power

¹¹Texas Instruments had pioneered the concept more than a decade before with their microprocessors TMS34010 and TMS34020, whose instruction set included graphics-specific functionalities [69].

¹²Fragment shading is the usual computer graphics terminology to designate the operation of deciding which colour each pixel in the screen will take. However, since further processing may be applied (notably antialiasing), those non-final pixels are referred to as fragments instead of pixels.

Figure 2.5: An early example of pipelining at the Ford factory, 1913. Each worker (functional unit) performs a different function and the assembly line as a whole processes many parts in parallel, although the manufacturing procedure of each part is strictly sequential. Admittedly, the analogy is not complete because, unlike in a microprocessor, each worker always does exactly the same action. A tray line in a canteen would probably be a better analogy but the photo would not look as nice.

consumption is paramount. In the other end of the scale, supercomputers are also very sensitive to energy efficiency because their electricity bills can be huge, a significant portion of them in refrigeration; today, many of the entrants, including some in the Top-10, of the TOP 500 list of the most powerful supercomputers in the world [2] feature GPUs as coprocessors.

2.3 Heterogeneous computing, or why CPUs and GPUs complement each other

Both in terms of raw floating-point performance (FLOPS) and of energy per floating-point instruction [3], GPUs have currently much better performance than (multi-core) CPUs (about $10\times$ [88] and $100\times$ [20], respectively), which has prompted developers of many scientific applications to write their applications to take advantage of these capabilities. Why do GPUs and CPUs offer such different performance? Do designers of GPUs really have an edge over their CPU colleagues? The obvious answer is a resounding no: they are just devices that respond to radically different design philosophies and each excels at those tasks it was designed to do.

On the one hand, CPUs come from a tradition of latency-oriented machines where, as mentioned in section §2.1, the paramount goal is maximum execution speed of general-purpose sequential code. Before the emergence of multi-core machines, the way to achieve this, other than simply turning the frequency up, was exploiting instruction-level parallelism: even though the processor is executing a sequential program, there are many functional units within the processor (instruction fetch, instruction decoding, arithmetic logic unit (ALU), etc.), so it makes sense to try to simultaneously use as many of them as possible.

The simplest way to do this—and raise operating frequency—is pipelining (see figure 2.5): the processor is divided into the various stages that make up the execution of an instruction

Figure 2.6: Informal comparison of transistor area dedicated to different functions in CPUs and GPUs. © NVIDIA Corp.

and in each clock cycle a new instruction is admitted into the first stage, while the previous instruction is at the second stage and so on. However, this means that either all the stages in the pipeline should have equal duration (this was one of the design goals of reduced instruction set computing (RISC) machines) or the slowest steps will stall the queue. In order to solve this problem, *superscalar* processors broke the linearity of the pipeline and introduced out-of-order execution (OOE): the processor fetches and decodes several instructions and analyses their interdependence; if it is feasible to simultaneously dispatch different instructions to different functional units of the processor without breaking their causality, the scheduler may choose to alter their sequencing with respect to the original order in which the compiler wrote them.

General-purpose sequential programs contain a lot of branching (around 20% [167]): depending on received inputs, different code needs to be executed. This poses a fundamental problem to pipelining: it is impossible to know which branch of code needs to be executed before a condition is evaluated and therefore the pipeline stalls until then. The standard solution, which is still a very active field of research, is branch prediction and speculative execution, whereby the processor tries to guess which will be the correct branch and preventively fetches and executes its instructions. The catch, of course, is that if the prediction is wrong not only execution resources will have been wasted but also the side effects of the wrongly executed branch need to be reversed. In order to support all these sophisticated techniques, CPUs need to devote a large portion of their area (transistors) to large caches and a lot of control circuitry that consume power but do not contribute to numerical performance.

On the other hand, GPUs come from the world of video games, where the amount of work is always massive: for each video frame, millions of floating-point operations must be performed and therefore designers of GPUs choose to cram as many floating-point arithmetic units as possible within the chip at the expense of other functions (see figure 2.6). Now, if they dispense with all the sophisticated circuitry that allows CPUs to execute their code so fast, should they not be much slower? They actually are! The end-to-end execution time of a single execution thread (latency) in a GPU is much higher than in a CPU for a given sequential code. These devices are said to be throughput-oriented, there are so many triangles and pixels to process that the latency associated to one single vertex or pixel is unimportant: it is necessary to have them all processed in order to show the next frame¹³, so the goal is to be able to process as many of them

¹³Frame periods are in the order of milliseconds, which is a very long time in processor scale, hence the relatively low importance of having low latency in a GPU, whereas a CPU needs to be more responsive. Latencies in the order of milliseconds can be awkward for keyboard or mouse interaction and simply unacceptable in control applications.

Figure 2.7: Latency- (left, Learjet 75 business jet) and throughput-oriented (right, Sovereign Mærsk “Susan Mærsk” container ship) means of transportation. Depending on the amount of (uniform) work to be done, one option will make more sense than the other. On the one hand, delivering a kidney to be transplanted in a cargo ship would be ludicrous because it is latency-sensitive, so a jet is a good choice; on the other hand, delivering by airplane all the cars produced by a factory could theoretically be faster than a ship, but a huge number of (costly) aircraft would be necessary and fuel costs would be astronomical. Of course, real-world logistics are *heterogeneous* and different means of transportation are selected for different stages of a chain from the factory to the final customer.

per second as possible. In order to increase throughput, GPUs use very deep pipelines and, most importantly, many processing cores.

The fact that GPU cores are relatively simple in comparison with CPU cores directly translates in a different model of parallelism from multi-core CPUs. Whereas single-core CPUs respond to the single instruction, single data (SISD) architecture¹⁴ defined in Flynn’s widely-used taxonomy [58] and multi-core CPUs respond to the multiple instruction, multiple data (MIMD) architecture—each of the processing cores can execute completely independent code (task parallelism)—, GPUs lack the control abilities to operate each of its cores independently. Instead, GPUs adhere to the single instruction, multiple data (SIMD) architecture, where all the cores are executing the same sequential code (which allows using even less chip area for flow control), each on different inputs and writing their outputs in different addresses. Furthermore, although all cores execute the same code, there is no guarantee whatsoever that they do so synchronously or even that two threads will execute in a particular order. Therefore, programs written for a GPU must be designed so that each output datum can be independently computed by a single execution thread from the input data without collaboration from any other execution thread¹⁵. Algorithms that respond to this requirement are said to be data parallel because the source of parallelism is in the nature of the data itself, rather than the fact that there are several different tasks to be done concurrently. Obviously, since it is the motivation behind architectural choices of GPUs, the graphics pipeline is an excellent example of a data-parallel problem because each 3D vertex is independently transformed and projected onto the screen and the colour of each pixel is computed independently of the colour of other pixels.

Computer vision is a prominent area where applications frequently exhibit data parallelism,

¹⁴Modern CPUs include some degree of small-scale SIMD provided by vector instructions such as MMX/SSE extensions to the x86 architecture or AltiVec in Power architecture among others.

¹⁵In general, communication among execution threads is not possible. However, some restricted forms of communication and synchronization are possible under the CUDA (and OpenCL) computing model, detailed in section §2.4.

at least at some of their initial stages [120], in which input images are processed to enable further steps that will extract relevant information by means of mechanisms such as feature point detection [134, 21], global shape descriptors [12], frequencial transformations [49], image filtering [128], etc. Most of these operations are independently performed pixel- or region-wise, which makes them amenable to parallelisation. In addition to image-specific algorithms, many computer vision algorithms also employ generic machine learning algorithms such as neural networks [36] or support vector machines [161], that are also good examples of data-parallel structures.

Data parallelism is interesting because it is inherently scalable: since each output datum is computed independently, as long as there are many more data elements than processors, the computation can be transparently sped up by increasing the number of processors. However, real world problems/algorithms are rarely completely data-parallel and fall somewhere between task- and data-parallel. Sometimes, careful examination of the problem can lead to a new algorithm that is [more] data-parallel, but most often there are parts of the algorithm that are inherently sequential. Therefore, as in the case of a logistics chain (figure 2.7), heterogeneous computing, i.e., a balanced mix of CPU and GPU, leads to the fastest and most efficient solution.

2.4 The GPGPU computing model

As soon as the computer graphics APIs allowed programmers to write their own shaders, some adventurous researchers envisioned and pioneered the use of GPUs to perform general-purpose computations they had not been designed for [91, 157] (and even before that for computational geometry problems [77]). The task was not easy because the only way to feed input data to the GPU was to upload textures and the only way to get results out of it was as pixels of a dynamically rendered texture. Furthermore, the algorithm had to be programmed in assembly and numerical accuracy was poor because video game graphics did not need it and therefore GPUs were not IEEE 754-compliant. However, they did manage to demonstrate the validity of this kind of architectures for general-purpose computation and, probably, showed hardware manufacturers an untapped market.

In 2007, shortly after the launch of their first unified shader architecture (that did support IEEE 754 floating-point arithmetic [119] and could access arbitrary addresses in memory), NVIDIA presented the CUDA (Compute Unified Device Architecture) programming language, which rather than a full-fledged language is a set of extensions to the C++¹⁶ programming language(s) that enabled programmers to write general-purpose programs to be executed on their GPUs without having to shoehorn them through the ill-suited graphics APIs¹⁷. In the following years, many other analogue languages or toolkits appeared, such as Khronos' OpenCL, AMD's Stream SDK, Microsoft's DirectCompute, Apple's Metal or Google's RenderScript. We have chosen to work with the CUDA programming language, although it is proprietary, because it is reasonably terse and is rapidly evolving to mirror new hardware capabilities offered by NVIDIA; standard languages such as OpenCL evolve much slower (and OpenCL in particular is quite cumbersome to use). Therefore, the CUDA terminology will be hereinafter employed to explain the core concepts of general-purpose computation on graphics processing units (GPGPU), which are common to all of these languages even if they use different terminology.

¹⁶Although C++ as a whole is far from being supported due to the inherent complexity of features such as dynamic polymorphism, most of the compile-time features of C++ are supported in CUDA, providing greater expressive power to developers.

¹⁷Independent researchers had already produced such a language [29] to abstract the (still underlying) graphics API, but without the advantage of manufacturing their own hardware, they could obviously not match the functionality that CUDA would bring.

Figure 2.8: Typical control/data flow of a GPGPU application: 1. Memory is allocated in the device to hold input and output data for the computation; 2. Input data is copied from the host memory to the device memory; 3. Computation runs on the device; 4. Output data is copied from the device memory to the host memory. The host can issue commands for steps 1–3 without blocking and perform other work while they execute. Picture courtesy of Sylvain Collange [38].

2.4.1 Program structure

Programs written in CUDA are very similar to a standard C program, which starts execution at the *main* function and progresses sequentially through it, calling subroutines as needed. However, in a CUDA program some of the functions are marked as *device* code, which means that, when compiled, the corresponding machine code is targeted to the GPU(s) instead of the CPU. In this execution model, GPUs (*computing* devices in CUDA terminology) are coprocessors to the CPU (*host*), and in most cases they have their own onboard memory. The host controls the main flow of execution of the program, manages memory transfers to and from the device and submits commands (work) to the GPU, as illustrated in figure 2.8. The GPU executes commands asynchronously, freeing the CPU to perform other tasks meanwhile.

The first consequence of this model of execution is that not everything is worth executing on the GPU, even if the structure of the problem is amenable to parallelisation: although the GPU has much higher floating-point throughput than the CPU, the overhead latencies associated to copying data onto the GPU may offset the gain of faster floating-point operation if the problem size is too small. In fact, as already mentioned in section §2.1, CPUs are already faster than their memory, which means that in order for the GPU to be faster than the CPU, the problem must be complex enough, or more specifically, have a reasonably high ratio of floating-point operations to memory accesses (arithmetic intensity) [65]. Alternatively, even if this is not the case, it may make sense to offload work to the GPU if there are other meaningful tasks for the CPU to do.

Functions to be executed on the device adhere to the stream processing paradigm, which is a specific form of SIMD¹⁸ designed to exploit data parallelism. Given a set of data or *stream*, a sequence of operations or *kernel function* is applied to every element in the stream. Thus, the CUDA application programmer only needs to write the kernel function and the runtime and hardware transparently¹⁹ issue, schedule and synchronize all the execution threads.

A simple way to look at it is to think of kernel functions as the body of loops that iterate over

¹⁸This programming style is also referred to as single program, multiple data (SPMD), depending on the level of abstraction it is looked at. Theoretically SIMD implies that every instruction is run in lockstep for all the data elements, which is not true for arbitrary sizes of data; SPMD is a less restrictive definition [6] that only implies [possibly asynchronous] execution of the same sequence of instructions on all the data elements.

¹⁹At least in a first approximation. In practice, the programmer needs to be aware of a number of implementation details in order to optimize performance.

Figure 2.9: The basic unit of the SIMT architecture, the streaming multiprocessor (SM). Note this particular figure illustrates the Fermi microarchitecture; the specific number of functional units of each kind may be (and actually is) different in other microarchitectures. © NVIDIA Corp.

all the elements of an array. For instance, in order to add two vectors²⁰ in a sequential program, a loop is used where the loop counter starts at 0 and ends at $n - 1$ and is added to the base pointers of all three operands, both input arrays and the output array, so that in each iteration i , the i -th element of the sum vector is computed. In a stream program, the kernel function is just the addition of two scalars and as many concurrent execution threads are instantiated as there are elements in the vectors. In this case, the identification between kernel functions and the body of a loop is straightforward, but the fact that stream processors run each “iteration” of the loop in concurrent threads has much deeper implications:

- Each iteration of the body of a loop can modify variables outside the scope of the loop, affecting the results of next iterations. In the stream processing paradigm, all the iterations of the loop may run concurrently and independently, so partial or final results from one thread cannot be used by another thread, in principle.
- The order in which elements are processed in a loop is well defined. However, in the stream processing paradigm, the hardware may schedule threads for execution in any order, therefore the correctness of the algorithm to be implemented cannot depend on a particular ordering of operations²¹.

These restrictions mean that many algorithms need to be thoroughly redesigned to be run on a GPU and some others have intrinsically sequential components that make them ineligible for parallelisation. In practice, as discussed in the next sections, the actual architecture of modern GPUs exhibits departures from this theoretical model that make a bit easier, or possible at all in other cases, to adapt a greater variety of algorithms to be run on them.

2.4.2 SIMT architecture and thread hierarchy

As mentioned before, GPUs sacrifice many of the microarchitecture advances of superscalar CPUs to maximize throughput and adhere somewhat loosely to the SPMD architecture. The specific implementation that NVIDIA (and other vendors, though details may vary) uses in their GPUs is known as single instruction, multiple threads (SIMT) and is designed to provide transparent data scalability across devices with different number of functional units.

The basic hardware unit of the SIMT architecture, pictured in figure 2.9, is known as streaming multiprocessor (SM). It features multiple scalar processing cores, each of them with an ALU and a FPU, multiple load-store units and multiple special function units (SFU, specialised floating-point units that implement transcendent functions in hardware). However, none of these functional units can operate of its own accord. Instead, there is one or more shared schedulers and dispatchers that read, decode, schedule and dispatch instructions to the appropriate functional units, that can and do operate simultaneously on different threads.

In order to simplify the design of the scheduler(s), threads are scheduled in vectors (as in vector processing) of 32 called *warps* (see figure 2.10 for the etymology of the term). This scheduling strategy does make much sense because, since all threads in a warp are doing the same, when they get stalled they do so together; when they need to read/write from/to memory

²⁰This example is hopefully good and simple enough to illustrate the concept of kernel function, but an exceedingly bad candidate to be implemented on a GPU; its arithmetic intensity is so low that it would never perform better on a GPU than on a CPU [65], at least if the immediate origin of data is the main memory of the host, because the data transfers take substantially more time than the computation itself.

²¹Although they are [incorrectly but] routinely thought of as real numbers, floating-point numbers do not (among other problems) satisfy the associative property [114]. Of course, this is also true in sequential programs, but parallelism only makes it worse because it may be difficult to establish a correct order for the operations to converge; recent proposals [68, 81] try to address this growing concern, but no hardware implementations exist yet.

Figure 2.10: The term *warp* comes from the field of weaving: it designs the parallel threads that are kept in tension on a loom so that other thread(s), called weft, can be inserted between them to weave a fabric. Picture courtesy of Madelyn van der Hoogt.

Figure 2.11: The scheduler switches to other warps while it waits on long-latency operations. As long as there are enough warps ready to be scheduled, latencies can be completely hidden. Picture courtesy of Kayvon Fatahalian [93].

Figure 2.12: Sequential execution of different branches taken by threads in the same warp. A single divergent thread is enough to force the scheduler to execute both branches, thus halving throughput. If there are more than two branches the problem worsens, until in the extreme case that all threads in a warp took different branches, throughput would be $1/32$ of that of sequential code. Picture courtesy of Kayvon Fatahalian [93].

they need it at the same time. Therefore, their execution can efficiently be suspended and resumed. In fact, execution threads in CUDA have their own registers assigned from beginning to end of the kernel function and they do not usually make use of stack frames²². This means that whenever they are suspended, there is no context nor stack to save and the scheduler can simply switch and dispatch instructions from other threads with zero overhead. Therefore, no sophisticated OOE circuitry or branch prediction is needed to keep all functional units of the SM occupied at all times as long as there is enough work to do (i.e., threads to execute), as illustrated in figure 2.11. Of course, the downside to threads having such small footprint is that the complexity of the code of a single kernel function is limited by the number of available registers²³. Thus, since there is a single big register file per SM, the more complex the kernel function, the fewer threads can be scheduled for execution in the SM. Therefore, it is sometimes advisable to decompose the problem into several simpler kernel functions rather than a single complex one in order to keep a high number of threads in flight.

Since it is the minimal unit of scheduling, all threads in a warp are guaranteed to execute each of the instructions in the program coherently, which can be used as a low-level synchronisation construct. However, kernel functions are allowed to use conditional code and it is obvious that threads in different branches cannot execute coherently. These apparently incompatible statements reconcile at the expense of throughput: whenever there is a path divergence between threads of the same warp, all branches need to be executed sequentially, selectively disabling cores mapped to threads that did not take the current branch, as illustrated in figure 2.12. Therefore,

²²Though calling nested functions is possible, it is discouraged because stack frames are allocated on off-chip memory and are therefore much slower.

²³In regular sequential programs on CPUs, there is also a limit to the complexity of the code determined by the size of the stack; however, non-buggy programs rarely need to recurse deep enough to cause a stack buffer overflow. However, if there is a legitimate need to use a bigger stack, most operating systems allow changing the size of the stack, which is hosted on a buffer of [abundant] regular memory anyway rather than on [scarce] hardware registers.

Figure 2.13: Hierarchical thread organisation in a 2D grid of 2D blocks to process an image. The image has been heavily subsampled so that individual threads can be seen, whereas in real cases there may be millions of threads to process. Since all blocks have the same size, the number of allocated threads may exceed the image size (blank squares in the illustration); these excess threads must be prevented from doing actual work in order not to corrupt memory.

branching is best used sparingly and some algorithms will not make sense in the GPU because of this limitation. However, if whole warps take different branches there is no performance penalty because only one branch is scheduled in each warp; fortunately, execution threads are organised hierarchically, which helps design algorithms that make efficient use of conditional code.

Threads are mapped to data under the stream processing model, and CUDA offers a natural way to do so, illustrated in figure 2.13. Threads are grouped into 1-, 2- or 3-dimensional *blocks*, and blocks are in turn organised into a 1-, 2- or 3-dimensional *grid*²⁴; the size of blocks can be chosen by the programmer to better suit the task at hand and the corresponding grid size is dynamically determined in run-time. Threads can determine their coordinates, both local within the block and global, in order to distinguish themselves from other threads and identify the appropriate portion of data they need to process.

The reason for establishing this hierarchy instead of a plain multidimensional array of threads is that, much as warps are the minimum scheduling unit within each SM, blocks are the minimum scheduling unit at the device level. Thus, a GPU can have any number of SMs to dispatch thread blocks to, but all threads in a block are always guaranteed to be processed by a single SM, which will allocate at once resources to process as many blocks as its shared resources allow. In a typical grid, there are many times more blocks than SMs, so they are sequentially dispatched to SMs as they become available in an unspecified order, providing transparent scalability with the same code: high-end devices have more SMs (and/or more SPs per SM) and can process a given dataset in fewer iterations while lower-end devices will need more iterations (see figure 2.14).

²⁴Formally, both blocks and grids are always three-dimensional, with the sizes along unused dimensions set to one.

Figure 2.14: Blockwise dispatching provides transparent scalability; GPUs with more SMs simply process more blocks per batch. © NVIDIA Corp.

Since all threads of a block are guaranteed to always be simultaneously in flight²⁵ on the same SM, the restrictions of a pure stream architecture can be relaxed: the order in which the threads execute is still unknown but it is easy for the warp scheduler to provide intra-block synchronisation barriers simply by not scheduling any warp from the block to execute past a specified instruction. Coupled with the ability of threads to write on any memory address, this mechanism enables limited intra-block communication between threads that can be used to implement a wider variety of algorithms. However, synchronisation barriers limit the number of eligible warps to schedule; since having numerous warps to schedule is the primary mechanism for latency hiding, careless use of synchronisation barriers may result in frequent pipeline stalls and low throughput.

2.4.3 Memory hierarchy

There are GPUs that share the main memory with the CPU, especially in mobile phones, while discrete GPUs typically feature a few GiB of onboard memory specially designed for high bandwidth (often $> 10\times$ than regular memory, again at the expense of higher latency). Even though the latter incur the penalty of having to copy data from/to the main memory of the host, as discussed in 2.4.1, they usually outperform otherwise comparable integrated GPUs by a sizable margin [147] because, since a large number of threads maps to a large dataset in the stream computing model, high bandwidth is necessary to access data.

However, even assuming that an application in a GPU is able to completely and efficiently saturate the memory bus, there is a clear mismatch between computing capabilities and memory bandwidth. For instance, the NVIDIA GTX 970 has a memory bandwidth of 224 GiB/s; this translates into ~ 57 G single-precision floating-point operands, which is a far cry from the theoretical computing throughput of this device, rated at 3494 GFLOPS; in other words: in order to keep all computing resources occupied, the algorithm must perform ~ 61 floating-

²⁵The execution of each thread (or, more precisely, the warp it belongs to) may be active or suspended at any moment, but all threads in the block have been instantiated at once.

Figure 2.15: Influence of memory access patterns in the effective bandwidth from global memory.

point operations per operand read from global memory (cf. ~ 7 floating-point operations for a contemporary CPU such as the Intel Core i7 4770K). How can an application then tap that computing power if memory cannot serve enough data to feed the processors? The answer lies in the fact that almost all non-trivial computations use each input datum more than once and, at any rate, need to perform several ancillary operations per input datum; consequently, if an algorithm does not exhibit these traits (in enough degree), it may not make sense to run it on a GPU.

The CUDA architecture provides a tiered memory model that helps both the compiler and the programmer to place data that will be used often closer to the processors, thus reducing the pressure on the main memory of the device. Specifically, there are three distinct storage categories, each with different scope and characteristics:

- Register file: with thread-local scope, it has a capacity of a few tens of 32-bit words per thread and stores the automatic scope variables of a kernel function. With a bandwidth around 40 TiB/s, operations whose operands come from registers can easily saturate the computing cores.
- Shared memory: located within each SM, it can be accessed concurrently by all threads in the same block. It has a capacity of a few tens of kiB per block and a respectable bandwidth around 1 TiB/s which makes it fairly easy to saturate the computing cores. It allows intra-block communication among threads of the same block and must be explicitly programmed for.
- Global memory: located off-chip, it can be accessed by every thread in any block or grid, even across different kernel functions. Its capacity is typically of a few GiB, but its bandwidth is relatively low, around 200 GiB/s.

It is therefore clear from this figures that a good memory access strategy is crucial to achieve good performance when writing applications for a GPU. Many applications exhibit a desirable trait known as locality of reference, which in this context means that neighbouring threads usually need to access the same or neighbouring data (spatial locality) at roughly the same time (temporal locality). The hardware architecture is designed to help exploit locality both through the scheduling model—grouping the execution of neighbouring threads in blocks enforces temporal locality to a degree—and through the provision of shared memory and level 1/level 2 caches to access global memory, which help absorb the small jitter derived from the warp-level, rather than block-level, coherent execution.

However, the burden of ensuring spatial locality falls entirely on the application designer, who is responsible for ordering operations, mapping execution threads to data and for arranging data in the most favourable layout to ensure that threads in the same warp access consecutive memory addresses so that the load/store units of the SM can coalesce their petitions into as few transactions as possible. The price to pay for not doing so, illustrated in figure 2.15, is steep indeed: memory transactions are at least 32 bytes in size, therefore scattered reads may result not only in greater latencies (all operands from all threads in a warp must be retrieved in order for it to be scheduled for execution) but also in a reduction in effective bandwidth (and reduced efficacy of the caches) because many irrelevant data are retrieved from memory but not used. Consequently, optimizing the usage of bandwidth is paramount and usually the aspect in the design of GPU applications that requires the most effort from the programmer.

2.5 Conclusion

The trends in computer architectural design run unequivocally towards ever increasing parallelism rather than more sequential processing speed. At the same time, the amount of information to process grows as digital cameras, smartphones and all kind of devices pervade our everyday life. Consequently, all new algorithms in all walks of computing will have to be designed with parallelism in mind or risk a premature obsolescence as the size of data sets continues to grow.

In the past few years, graphics cards have evolved into a new class of general-purpose massively parallel computing devices that have very high computing throughput and energy efficiency. These devices are especially well-adapted to problems exhibiting data parallelism, such as image and video processing or many computer vision algorithms. Although the technology is relatively new and their architecture and development tools are still rapidly evolving, it has reached a usable stage that enables developers to address bigger problems or achieve real-time operation in others at moderate cost.

Although the relatively slow throughput of memory is a serious threat to the development of CPUs and GPUs alike, the next years will undoubtedly bring further integration of GPUs as standard elements of all computers in a virtuous development circle in which more and more algorithms will be developed and ported to these devices, and future architectural changes in these will be in turn influenced by the needs of those algorithms.

Chapter 3

Evaluation of Backward Mapping DIBR for FVV applications

3.1 Introduction

In the past few years the field of free viewpoint video (FVV) has seen much interest, both due to the emergence of 3D cinema and TV (as a means to generate the several views that these systems need), or as a new creative tool to obtain so called “bullet-time” effects such as those of the famous scene of the sci-fi movie *The Matrix*. However, FVV is a term that describes a goal rather than a specific way to achieve it. Actually, there are several technologies that serve this purpose with varying degrees of success depending on the constraints of the specific application. Methods for achieving FVV range from the obtention of an explicit model based on geometric meshes or voxels [105, 133] to elementary manipulation of pixels from source images with varying degrees of knowledge with respect to where those images were taken from.

The former class of methods offers great flexibility for moving the camera because once the model has been obtained, rendering new views turns into standard computer graphics procedure. However, this flexibility comes at a price: there is no known general method to obtain high-quality geometric models of an unspecified scene and, even in those kinds of scenes for which models can be obtained [31, 124, 125], reconstruction methods are computationally expensive.

On the opposite side of the spectrum, there are methods in which every pixel in the virtual image results from a combination of contributions from pixels in the original images and no attempt is made at understanding the semantic structure of the 3D scene. These methods are collectively known as image-based rendering (IBR) or depth-image-based rendering (DIBR) if additional depth information is available.

The fundamental idea of DIBR algorithms [109, 56] is that, having images from one or more cameras (and viewpoints) containing both color and depth information, each pixel is easily back-projected to a 3D point, all pixels together forming a 3D point cloud. Then, this point cloud can be projected onto a new virtual camera in order to obtain an approximation of the image that a real camera with the same configuration would have seen. This series of operations involves several matrix multiplications per input pixel (more details in section 3.2.1) and additional image filtering operations; therefore it carries significant computational cost. Fortunately, the operations that must be applied to each pixel are largely independent and therefore this application is a prime candidate for GPU implementation, detailed in section §3.4. However, other than its cost, there are several problems with DIBR synthesis that must be dealt with when mapping

images to interpolate the new view:

- When looking at the 3D scene from a different viewpoint, some parts of it that were occluded from a reference viewpoint should become visible. This is a fundamental problem that can only be objectively solved by placing enough reference cameras so that no area of interest is left uncovered. However, methods have been described for guessing how to cover those areas in a subjectively acceptable way [102, 155].
- Each pixel from the original image(s) maps to at most (due to occlusions) one pixel in the target image. In general the coordinates of each 3D point, when projected onto the target image space, are non-integer, so some resampling is necessary. The fastest and therefore usual solution is to map the contribution of that point to the nearest neighbour, which usually results in cracks in the image. These effects can be corrected by interpolating neighboring pixels or using splatting techniques (i.e. covering more than one pixel in the target image per 3D point). However, all of these techniques amount to low-pass filtering the image, thereby imposing a penalty on sharpness. In the end, these are patches for what is a well known problem in the image-processing domain: in order to completely cover the target domain, geometric transformations must be performed backwards from the target to the reference image(s). The idea in this context was already proposed by [140] and has also been implemented for the MPEG View Synthesis Reference Software [116, 158] and refined in more sophisticated methods [79].
- Pixels on the edges of objects have colours that are often combinations of foreground and background objects. However, it makes no sense to average depth information: it must be set to a definite value, either foreground or background. Consequently, when assigning that mixed colour information either to the foreground object or to the background, some strange halo or ghosting artifacts are produced. Some proposals have been made to mitigate this effect, such as perform splatting only for the contributions coming from edge pixels [158] or layering contributions coming from foreground, background or edges and prioritising them in order not to employ edge contributions if not absolutely necessary [118, 169].

Most of the DIBR algorithms that have been presented in the literature in the past few years are intended for use in 3D TV applications [152, 118, 112]. As such, their main concern is to generate views appropriate for several viewers who are sitting in front of a TV screen and look at the scene from only slightly different angles and roughly from the same horizontal plane. Following these contour conditions, they care mainly about situations in which the cameras are disposed in a 1D array, or can be reduced to that setup using the camera calibration parameters and simple image warping procedures. In fact, this configuration has been subject of special interest by the MPEG Video Group [158, 30] and has been shown to significantly reduce computational complexity by constraining all warps and interpolations to 1D operations [56].

If both source cameras and virtual camera share a common camera plane and their baseline distance is small, displacements of the virtual camera along the baseline do not usually result in significant changes of the apparent size of objects on the image plane. In this specific scenario, computational resources can be saved by using a simple forward mapping approach, that in this case may work well enough and does not usually result in very large cracks, although it usually requires some light postprocessing of the final image to work out whatever small holes might have been left uncovered.

If the baseline distance increases too much, though, parallel cameras cannot simultaneously see objects that are located near them and they must point in different directions in order to see the same objects. In this scenario, apparent sizes of both foreground objects and background can vary significantly with the choice of the virtual camera location. Specifically, if the apparent

Figure 3.1: Basic pinhole camera model placed at the origin of coordinates. © Cambridge University Press [72].

size of any element in the images increases in the virtual view, large cracks appear in the final image that are more difficult to fix.

However, we submit that DIBR is adequate not only for 3D TV applications but also for general FVV applications, even with camera configurations that do not lend themselves to the usual 1D reduction, this is, cameras widely spaced apart with their main axes describing big angles. However, care must be exercised when handling data: in particular, we found that some widely-used datasets [169] choose to relate all depth information to a global coordinate system instead of relating each depth video to the local reference system of the corresponding camera. We will show how this affects large baseline setups and why we think it is not a good choice.

3.2 Study of a wide baseline setup

We wanted to explore whether it is possible to correctly reconstruct the main characters in a scene using widely spaced cameras with DIBR techniques. However, most of the available sequences are from parallel camera setups and therefore not suitable for this purpose. We selected the well-known sequences *Ballet* and *Breakdancers* [169], that are made up of eight views per scene, each of them carrying both colour and depth information¹. These sequences have been used in numerous papers [110, 116, 122, 117], but usually pairs of cameras lying next to each other are selected. Instead, we will employ the outermost cameras of both datasets for our study.

3.2.1 Geometric foundation of DIBR

In order to better understand the DIBR rendering process, let us review the geometry of a pinhole camera model [72]. A camera is basically a device that transforms points in the 3D space into their central (2D) projections on a plane where the sensor is located. In the most basic camera model, only the focal distance f from the camera centre C to the image plane needs to be specified, so that the 3D point $\tilde{\mathbf{X}} = (X, Y, Z)^\top$ is on the same ray as the point $(fX/Z, fY/Z, f)$ that lies on the image plane; therefore, the projected 2D point in the coordinate system of the plane, pictured in figure 3.1, is $\tilde{\mathbf{x}} = (fX/Z, fY/Z)^\top$, just by disregarding the third coordinate of the coincident 3D point.

If both the 3D and 2D points are expressed using homogeneous coordinates, the projection

¹These sequences can be downloaded from <http://research.microsoft.com/en-us/downloads/5e4675af-03f4-4b16-b3bc-a85c5bafb21d/>

Figure 3.2: Image (x, y) and camera (x_{cam}, y_{cam}) coordinate systems. © Cambridge University Press [72].

can be written as a matrix-vector multiplication:

$$\begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{pmatrix} f & 0 \\ f & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}, \quad (3.1)$$

where equality is up to a non-zero scale factor because we are dealing with homogeneous coordinates. Throughout this chapter, we denote Euclidean (non-homogeneous) coordinate vectors with a tilde (e.g. $\tilde{\mathbf{C}}$), and their homogeneous equivalent with the same letter but without tilde ($\mathbf{C}$). To convert a vector from Euclidean to homogeneous coordinates, we just append an extra element set to one and to perform the inverse conversion, we scale the homogeneous vector so that the last element equals one, then keep all but the last element.

The projection in equation (3.1) assumes that the 2D origin of coordinates in the image plane is located at the principal point, where the principal axis meets the image plane. However, real cameras have a rectangular sensor whose center roughly corresponds to the principal point, and it is customary to place the origin of coordinates at the lower left corner of the sensor, as pictured in figure 3.2. In addition, and considering that there is a scale factor that is lost upon projection (an object at a given distance will produce the same image that an object double its size at twice the distance), it is usually more convenient to refer to points in the image plane in units that bear some intrinsic significance, most often pixel units, rather than in arbitrary units. Thus, the projection matrix K , also referred to as the intrinsic matrix (or intrinsic parameters) of the camera, is usually defined as

$$K = \begin{pmatrix} \alpha_x & 0 & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.2)$$

where α_x and α_y are the focal lengths of the camera expressed in pixel units² and (x_0, y_0) are the coordinates, also in pixel units, of the principal point in the image reference system. Therefore, the projected point, in pixel coordinates, can be written as

²Spatial sampling rate is not necessarily the same in the vertical and horizontal directions in real cameras (i.e., the pixels can be rectangular rather than square). Thus, $\alpha_x = fm_x$, where f is the focal length in world units, and m_x is the number of pixels per world unit in the x direction, and the same goes for the y direction. In this definition of K we are also assuming that pixels of the camera are aligned with the axes.

Figure 3.3: Transformation between world and camera reference frames. © Cambridge University Press [72].

$$\mathbf{x} = K [I_{3 \times 3} | \mathbf{0}_{3 \times 1}] \mathbf{X}. \quad (3.3)$$

The projection matrix allows us to project points onto a camera that is located at the origin of the world coordinate system. However, it is obvious that if we are to use more than one camera we cannot place them all at the origin of coordinates. The practical solution is to refer all 3D points to a common coordinate system, usually known as world coordinate system and transform them into each local camera coordinate system to project them (see figure 3.3). Thus, if $\tilde{\mathbf{X}}_w$ are the Euclidean coordinates of a point, the coordinates of the point in the camera system are $\tilde{\mathbf{X}}_c = R (\tilde{\mathbf{X}}_w - \tilde{\mathbf{C}})$, where R is a rotation matrix that describes the orientation of the (orthonormal) axes of the camera coordinate system in world coordinates and $\tilde{\mathbf{C}}$ is the position of the camera centre in world coordinates. It is possible to encode this geometric transformation into a single matrix-vector product if the 3D points are expressed in homogeneous coordinates:

$$\mathbf{X}_c = \begin{pmatrix} R & -R\tilde{\mathbf{C}} \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} \mathbf{X}_w. \quad (3.4)$$

Sometimes the camera centre $\tilde{\mathbf{C}}$ is not given explicitly and $\tilde{\mathbf{t}} = -R\tilde{\mathbf{C}}$ is used instead, which represents the position of the world origin in camera coordinates. Together, R and $\tilde{\mathbf{t}}$ (or R and $\tilde{\mathbf{C}}$, depending on the convention) are known as the extrinsic parameters of the camera. Thus, the full world-to-image transformation is given by

$$\mathbf{x} = K [R | \tilde{\mathbf{t}}] \mathbf{X}, \quad (3.5)$$

where $K [R | \tilde{\mathbf{t}}]$ is usually known as the projection matrix P .

3.2.1.1 Undoing the projection

Now, the basic idea of DIBR is that it is possible to interpolate new views of a scene by “unprojecting” points of an image to their original 3D locations, so that they can be projected again onto a new virtual camera. However, image points $\mathbf{x}$ by themselves are not enough because, as illustrated in figure 3.1, all the points that lay in the line that joins $\mathbf{x}$ to the centre of projection

Figure 3.4: Forward mapping of *Ballet*, note cracks on the foreground. Black regions correspond areas not covered by either reference camera.

have the same projection. Therefore, it is necessary to have depth information for each 2D point to uniquely determine the corresponding 3D point along the ray. Thus, if the depth image contains the Z coordinates of the points in the camera coordinate system, it is easy to obtain the corresponding 3D points:

$$\tilde{\mathbf{X}}_c = sK^{-1}\mathbf{x}, \quad (3.6)$$

where s is a scalar chosen so that the last element of $\tilde{\mathbf{X}}_c$ equals the known Z camera coordinate Z_c of the 3D point.

If the depth image is referred to world coordinates instead, the process is slightly different:

$$\tilde{\mathbf{X}}_w = sR^\top K^{-1}\mathbf{x} + \tilde{\mathbf{C}}, \quad (3.7)$$

where s is a scalar chosen so that the last element of $\tilde{\mathbf{X}}_w$ equals the known Z world coordinate Z_w of the 3D point.

3.2.2 Forward mapping

Forward mapping (or warping) is the simplest way to implement DIBR: for each pixel in each of the reference images, the corresponding 3D point is computed and then projected onto the virtual camera, painting the target pixel with the colour of the original pixel. Unfortunately, as mentioned in section §3.1, the coordinates of the point in the coordinate system of the new camera will not in general be integer; therefore, the pixel³ with the closest [integer] coordinates will be painted.

In figure 3.4, a forward mapping rendition without postprocessing of the scene as seen from the midpoint between cameras 0 and 7 (see [169] for details on the configuration) is shown. Since the cameras have different fields of view, there are areas at the left and right of the scene where only one of the cameras provides information, accounting for the poor(er) quality of the image in those areas. However, there are also very noticeable cracks on the main characters that we aim to solve. To do so, we have used a backward mapping approach similar to that described in [79, 116], although instead of bilateral filters we have designed a custom filtering scheme based on median filters; figure 3.5 shows a summary of the process.

³Or pixels, if using a splatting approach.

Figure 3.5: Block diagram of the process pipeline.

 Figure 3.6: Z-fighting in the reprojection. Points A and B project onto the same point on the virtual camera, but B must prevail because it is closer in the reference frame of that camera.

3.2.3 Backward mapping

The key idea to backward mapping is that, if a dense depth map can be estimated for the virtual camera, every target pixel can be traced to a single 3D point that can, in turn, be projected back to one or more of the reference images to compute the final colour of the target pixel. Even though unprojected 3D points are likely to project onto floating point coordinates in the original images, these are dense and allow for interpolation, thereby eliminating the cracks. In order to estimate this virtual depth map, the same principles of forward mapping are followed. Each of the pixels in the reference images is unprojected to a 3D point, forming a cloud that is then projected onto the virtual view, only now we do not care about colours and we just store the Z_c coordinate of each visible point in the cloud.

It is worth noticing that, due to motion parallax, occlusions will appear and, therefore, not every point will contribute to the virtual depth map. The solution to determine visibility is well-known in computer graphics: a Z-buffer. When projecting each 3D point onto this buffer, a simple test must be done: if the Z coordinate of this 3D point is less than the value previously stored in this pixel, it replaces this value.

This process is illustrated in figure 3.6. Regardless of the order in which A and B (unprojections of a' and b' from the original camera) are projected onto b'' , if the closest value is stored, in the end the pixel mapped to b'' should contain the Z coordinate of B , since A is not visible from the virtual camera. This also means that Z coordinates stored in the virtual depth buffer should relate to the reference system local to that camera (indicated by Z_v in the figure) instead of the world reference system (Z_w in the figure), although original data from the cameras may be all expressed in terms of world Z . It can clearly be seen on the figure that in world coordinates, A 's Z coordinate is less than B 's, but A should not be seen from *Virtual*, so world coordinates must not be used for determining point visibility.

Figure 3.7: Depth map estimation on the virtual camera obtained from only one reference camera.

Figure 3.8: Depth map estimation on the virtual camera obtained from only one reference camera after a filtering stage (cf. figure 3.8).

At any rate, since this operation is indeed a forward mapping, this virtual depth map (see figure 3.7) exhibits cracks just the same as colour forward mappings do.

However, there is a crucial difference between depth and colour forward mapping: whereas it is very difficult to tell whether a particular pixel in a colour buffer is correct or not (e.g., how to automatically distinguish a crack from an actual striped texture?), this buffer actually holds a description of the physical structure of the scene; this allows us to make several useful assumptions: real-world objects are usually connected and, barring grids, nets and strings (and actual cracks in the wall!), there are not many objects that could fit the physical description that cracks in the depth image imply. Therefore, it is relatively safe to apply 2D filtering techniques such as median filters, which would eliminate cracks and smooth the surfaces while retaining sharpness in the actual boundaries of objects.

3.2.4 Filtering depth maps

In order to close the holes in the depth maps, we selected a 5×5 median filter, which is known to perform in a way which is similar to a low-pass filter in smooth areas while retaining sharp contours. However, using the assumptions discussed earlier we can improve upon the basic filter. Since reference depth data is bounded and we know the calibration parameters of the cameras, we can surely know in which range to expect legitimate depth values in the depth buffer of the virtual camera. We previously initialise the Z-buffer on the virtual camera to a huge value well beyond expected depth values to signal a non-valid pixel. This allows us to make the following modifications to the median filter:

- Instead of using all values in the window, we compute the median of only the valid samples within the window, disregarding pixels known to contain invalid depth values.
- To prevent spurious isolated points completely surrounded by invalid points from growing, which would be an undesired consequence of the preceding modification, we require at least one quarter (experimental value) of the pixels in the window to be valid to actually modify the value of the pixel in the Z-buffer.
- To prevent valid depth values from *creeping* into areas that rightfully contain invalid values because they represent actual occlusions, we require at least half of the pixels in the perimeter of the mask to hold valid values. Thus, we can close small holes but not affect large blanks, which are probably correct.

The results of this filtering stage can be seen in figure 3.8. All the cracks have all but disappeared, and the remaining holes are mainly due to occlusions, either real or due to noise in the reference depth estimation. In our particular setup, we have the contributions of two cameras as inputs, so we have to merge the depth maps resulting of their reprojections. Since we have only two, there is no objective way to tell which one is right when they hold different values for the same pixel (a possible solution to this issue has been proposed to enhance results in boundary areas by examining the neighbourhood of each pixel and applying some inference rules [166]). Thus, in such a case, we just take the lesser value, because it could represent an object which is not visible from the other camera and that, if looked at from the new virtual viewpoint, occludes whatever was.

Figure 3.9 shows the final depth estimation obtained from the combination of the contributions of both reference cameras as described plus another additional filtering step to fill small holes. Most of the artifacts due to the forward mapping process have disappeared, and only those areas which are not seen from any of the cameras are missing information. While the quality and coverage of the depth map could surely be improved by adding information from more cameras, it is actually good enough to reconstruct at least the main characters in the scene.

3.2.5 Pixel colouring

Once a dense depth map has been obtained, all that is left is tracing back each pixel to the appropriate coordinates in the original images to get its colour. However, not all pixels in the virtual camera come from points seen by both cameras; we need to evaluate in a per-pixel basis to which camera(s) to refer to compute the final colour. To solve these occlusion problems, we have followed a simple strategy. During the virtual depth map estimation phase, we had already produced separate partial depth maps from each reference camera such as the one shown in figure 3.8, so we can use them to determine whether each pixel should be traced back to a particular camera.

Figure 3.9: Final depth map estimation on the virtual camera obtained from both reference cameras (cf. figure 3.8).

Figure 3.10: Z masking. When the surface contributed by one of the cameras is clearly closer than the other, its colour from the former will be used; when both surfaces are close to one another, the colour will be computed from both cameras.

Figure 3.10 illustrates a cross-section of the depth map estimation process; the dashed line denotes the right camera contribution and the dotted line denotes the left camera contribution. There are three possible cases:

- Angular sectors that are only seen by one camera ($\widehat{AOB}$ and $\widehat{EOF}$). We shall obviously not trace those points back to the other camera.
- Angular sectors seen by both cameras but in which the contribution of one of them is clearly closer than the other ($\widehat{BOC}$ and $\widehat{DOE}$). We disregard the camera with the farthest contribution for colour mapping because it is clearly occluded.
- Angular sectors seen by both cameras and at a comparable depth ($\widehat{COD}$). We cannot reliably rule out either of the cameras, so we will combine their contributions to compute the final colour.

In this latter case, its colour is determined using a weighted linear combination from the original (interpolated) colours. It seems natural that, if the virtual camera is placed exactly where one of the reference cameras was, the interpolated image should be exactly as the reference image. Additionally, the same point seen from different angles and cameras may present different colour, so we would like a natural transition between them. Therefore, we apply a view-dependent scheme that assigns the non-normalised weight $w_{\mathbf{X},i}$ to the colour contribution from camera i for the point $\mathbf{X}$:

$$w_{\mathbf{X},i} = \begin{cases} float_{max}, & \text{if } f_i(\mathbf{X}) = 1 \\ \frac{f_i(\mathbf{X})}{1-f_i(\mathbf{X})}, & \text{if } f_i(\mathbf{X}) < 1 \end{cases}, \quad (3.8)$$

$$f_i(\mathbf{X}) = \frac{1}{2} + \frac{(\tilde{\mathbf{C}}_i - \tilde{\mathbf{X}}) \cdot (\tilde{\mathbf{C}}_v - \tilde{\mathbf{X}})}{2 \|\tilde{\mathbf{C}}_i - \tilde{\mathbf{X}}\| \|\tilde{\mathbf{C}}_v - \tilde{\mathbf{X}}\|}, \quad (3.9)$$

where $\tilde{\mathbf{C}}_i$ is the position of reference camera i , $\tilde{\mathbf{C}}_v$ is the position of the virtual camera and $float_{max}$ is the largest available number in the machine as a practical approximation of infinity, so that only camera i is considered if $\tilde{\mathbf{C}}_i = \tilde{\mathbf{C}}_v$. An alternative, and possibly more intuitive definition, is $w_{\mathbf{X},i} = \cot^2(\theta_{\mathbf{X},i}/2)$, where $\theta_{\mathbf{X},i}$ is the angle between viewing rays from virtual and reference camera i to point $\mathbf{X}$.

The results of the whole backward mapping process can be seen in figure 3.11. Excluding areas with too much noise (e.g. the floor) or that are occluded from both reference cameras, the macroscopic geometry of the scene is correct. In particular, the main characters are well reconstructed and completely defined despite the significant perspective change.

3.2.6 Interpolation of camera parameters

Since we are using just two cameras, we can only expect good results when moving along a path connecting them. If we stray outside that path, many disocclusions, which will be unsolvable with the available data, will occur. However, defining the path that connects the cameras is tricky. We had the requisite of going from exactly the view of camera A to that of the camera B and, if possible, extrapolating outside to test the limits of the reconstruction based on the information from these two cameras exclusively.

We have reviewed two previous algorithms for going from A to B. The simplest one consists in just a linear interpolation, element-wise, of both intrinsic and extrinsic matrices. This results in a straight path from A to B and obviously produces exactly the original parameters at the

Figure 3.11: Final results of backward mapping on sequences *Ballet* and *Breakdancers*, using the same parameters as camera 4 of the original setup.

locations corresponding to those of the original cameras. However, in the extrinsic matrix, the rotation sub-matrix is not really a proper rotation matrix and therefore distorts the location of points in 3D space, especially so when extrapolating beyond A or B.

The second algorithm consists of a linear interpolation for the intrinsic matrix and the location vector, then a spherical linear interpolation [144] for each of the column vectors in the rotation sub-matrix. This procedure preserves the orthonormality of the rotated basis, but only does so between A and B. Strange distortions appear when extrapolating beyond the position of the original cameras.

If we assume that both cameras converge on the main subject(s) of the scene, maybe connecting them in a straight line is not the best possible path. If the cameras are placed forming a wide angle from the subject, a straight line will result in considerable variation of the distance from the virtual camera to the subject, whereas it would be probably more sensible to circle the subject from as constant a distance as possible. We have therefore designed a general path building method that keeps a reasonable distance to the subject, does not induce spatial distortions and can be sensibly extrapolated beyond A and B.

We wanted to parameterise the trajectory with a single parameter λ valued ± 0.5 at the exact original camera configurations. To simplify the discussion, let us assume that cameras A and B actually point exactly to the same target 3D point, allowing us to define a plane in space in which the entire path and the target are contained.

Since we would like to be able to peek at the target from beyond the positions of the original cameras, our ideal trajectory would be a circumference with its centre on the target and passing through the original locations of the original cameras. However, in general the distance from each of the cameras to the target will not be the same, so the next best curve that fits the requirements is an ellipse.

We can easily parameterise the position of the camera $\mathbf{C}(\lambda)$ on an ellipse (technically $\tilde{\mathbf{C}}(\lambda)$, but we dispense with the notation and understand all vectors through the end of this subsection to be Euclidean) with centre on the target ($\mathbf{T}$), $\mathbf{C}_a$ and $\mathbf{C}_b$ being the positions of the original cameras:

$$\mathbf{C}'(\lambda) = \mathbf{T} + (\mathbf{C}_a - \mathbf{T}) \cos\left(\frac{\pi(2\lambda + 1)}{4}\right) + (\mathbf{C}_b - \mathbf{T}) \sin\left(\frac{\pi(2\lambda + 1)}{4}\right). \quad (3.10)$$

Figure 3.12: Top view of different trajectories around a target object. In red, the trajectory described by equation (3.10) varies too much its distance to the target; in green, that described by equation (3.11) maintains a more uniform distance.

This function is the ellipse of which $\mathbf{C}_a - \mathbf{T}$ and $\mathbf{C}_b - \mathbf{T}$ are conjugate radii, shown in red in figure 3.12. This path between $\mathbf{C}_a$ and $\mathbf{C}_b$ is not satisfying because it goes too far away from the subject if the cameras are looking from a close angle and it also gets too near the subject when extrapolating. However, the angular parameterisation is convenient. If we impose the additional condition that the distance from camera to target must always be between those of the original cameras, we obtain the orbit shown in green, described by:

$$\mathbf{C}(\lambda) = \mathbf{T} + \frac{\mathbf{C}'(\lambda) - \mathbf{T}}{\|\mathbf{C}'(\lambda) - \mathbf{T}\|} \cdot \left(\frac{\|\mathbf{C}_a - \mathbf{T}\| (1 - \sin(\pi\lambda))}{2} - \frac{\|\mathbf{C}_b - \mathbf{T}\| (1 + \sin(\pi\lambda))}{2} \right). \quad (3.11)$$

In the real world, however, cameras do rarely converge, so there is no single $\mathbf{T}$ to consider. We can, however, find the shortest segment that connects the lines on which the local Z axes from both cameras are located. If we denote the ends of that segment as $\mathbf{T}_a$ and $\mathbf{T}_b$, we can define $\mathbf{T}(\lambda)$ as a moving target so that when the camera is located at the position of either of the original cameras, it points in the same direction the original camera did:

$$\mathbf{T}(\lambda) = \mathbf{T}_a \frac{1 - \sin(\pi\lambda)}{2} - \mathbf{T}_b \frac{1 + \sin(\pi\lambda)}{2}, \quad (3.12)$$

so that we can define the local Z axis of the virtual camera c as

$$\mathbf{Z}_c(\lambda) = \frac{\mathbf{T}(\lambda) - \mathbf{C}(\lambda)}{\|\mathbf{T}(\lambda) - \mathbf{C}(\lambda)\|} \quad (3.13)$$

Once position and target are determined, there is only another degree of freedom to fix, which is camera roll. Again, we compute the closest points between the lines containing the local Y axes of the original cameras, $\mathbf{U}_a$ and $\mathbf{U}_b$, and substitute them for $\mathbf{T}_a$ and $\mathbf{T}_b$ in equation (3.12) to determine a moving *up*⁴ target $\mathbf{U}(\lambda)$. There is no guarantee, though, that $\mathbf{U}_a$ and $\mathbf{U}_b$ are located in the positive $\mathbf{Y}_a$ and $\mathbf{Y}_b$ semiaxes (which we assume to be oriented in roughly the same direction so that $0 < \mathbf{Y}_a \cdot \mathbf{Y}_b \approx 1$), so we could end up with the camera roll 180° off if we unconditionally used $\mathbf{U}(\lambda) - \mathbf{C}(\lambda)$ as the direction of the new local Y axis. Furthermore, in general $\mathbf{T}(\lambda) - \mathbf{C}(\lambda) \not\perp \mathbf{U}(\lambda) - \mathbf{C}(\lambda)$ and therefore we cannot strictly use $\mathbf{U}(\lambda) - \mathbf{C}(\lambda)$ as the

⁴Actually, in a right-handed coordinate system the local Y axis points downwards, but this does not affect the reasoning.

Figure 3.13: Detail of foreground texture registration errors. Left to right: wide baseline unfiltered forward mapping, wide baseline backward mapping (cameras 0 and 7 of the dataset), small baseline backward mapping (cameras 3 and 4 of the dataset).

direction of the new local Y axis, although it does capture the camera roll. However, we can use it to determine the local X axis:

$$\mathbf{X}_c(\lambda) = \frac{(\mathbf{U}(\lambda) - \mathbf{C}(\lambda)) \times \mathbf{Z}_c(\lambda)}{\|(\mathbf{U}(\lambda) - \mathbf{C}(\lambda)) \times \mathbf{Z}_c(\lambda)\|} \text{sign}((\mathbf{U}_a - \mathbf{C}_a) \cdot \mathbf{Y}_a). \quad (3.14)$$

Finally, since $\mathbf{X}_c(\lambda)$ and $\mathbf{Z}_c(\lambda)$ are orthonormal, we can easily define $\mathbf{Y}_c(\lambda) = \mathbf{Z}_c(\lambda) \times \mathbf{X}_c(\lambda)$, thereby finishing the definition of the extrinsic camera parameters; as for the intrinsic parameters, it is perfectly possible to use any intrinsic matrix K , but if we want them to be exactly equal to those of the original cameras when $\mathbf{C}(\lambda)$ matches their original positions, we can easily do so:

$$K(\lambda) = K_a \frac{1 - \sin(\pi\lambda)}{2} - K_b \frac{1 + \sin(\pi\lambda)}{2}. \quad (3.15)$$

3.3 Representation of depth data

Although the wide camera configuration performs well for the reconstruction of the geometry of the main characters throughout the whole path between cameras, there are very noticeable registration errors between the texture contributions from the reference cameras. Figure 3.13 shows a detail comparing forward and backward mapping in a wide baseline configuration, and backward mapping in a short baseline configuration; both methods show similar alignment problems in the wide baseline configuration, even though forward mapping data are not filtered in any way. On the other hand, the short baseline configuration does not exhibit these symptoms, so it would appear that there is a trade-off between freedom of movement and image quality, and indeed there is if the depth reference data are not high quality or are misrepresented. In this section we will show that the errors are not due to the camera configuration but to the reference data in themselves.

The datasets used for this study provide depth images for each of the eight cameras of the setup. However, the depth data is related to the world coordinate system (which is fixed to the local coordinate system of camera 4) for all of them, which means that all pixels with the same grey level represent points on a plane $Z_w = k$. Being these data quantized to fit into 8-bit integers, the choice of a common reference system results in unexpected side effects.

Figure 3.14 shows a 2D analogue of the problem; we want to describe the circumference using both cameras and a common reference system fixed to the local system of camera A. When the

Figure 3.14: Side effects of Z quantization: cameras that point in a direction that describes a large angle with respect to the axis depth is referred to encode severely distorted surfaces.

Figure 3.15: Left: original depth estimation; right: virtual depth reconstruction, both for camera 4. Note that the original depth is *not* a ground truth.

camera is aligned with Z (camera A), the information that is actually being encoded (only the lower half depicted) is reasonably faithful to the original object. However, when the camera is severely off-axis (camera B), what is actually being encoded significantly differs from the original surface. If data for the reconstruction comes from two cameras that lie on opposite sides of the neutral alignment, as it is the case in our study, both are biased towards opposite directions, so the reconstructed geometry is painfully wrong. Note that depth quantization in this figure is uniform for the sake of simplicity, whereas it is non-linear in the *Ballet* and *Breakdancers* datasets, but this does not affect the reasoning above.

Figure 3.15 shows a comparison between the original depth data estimation for camera 4—the only reference camera which provides local depth data and therefore the only one for which this comparison makes sense—of the *Breakdancers* dataset and the reconstruction of the virtual depth from the same viewpoint using the outermost cameras (0 and 7) as sources. Inconsistencies and errors in the depth estimation of reference cameras are augmented by the choice of a common reference system, resulting in the foreground object being much bigger than it should. However,

Figure 3.16: Unfiltered depth map estimation on the virtual camera obtained from a severely off-axis camera with Z quantization, final results of backward mapping for the same configuration.

it must be noted that the depth estimation of camera 4 is *not* a ground truth, so this comparison is merely illustrative.

These effects are not apparent when using pairs of cameras with a short baseline, even if they point at a moderate angle from the global Z axis. In such a case, quantization effects affect both cameras in a similar way and even if the geometric information they are encoding is wrong, it is consistently so. However, using closely placed cameras as sources severely hampers the freedom in “free viewpoint video”, whereas at a large scale the results from the camera configuration in our study are generally satisfactory.

Therefore, in order to test whether the failures exhibited by the extreme camera configuration are fully imputable to the effects just described or there is something intrinsically wrong about this configuration, we implemented a tool to export synthetic scenes from Blender as images plus depth, either relative to the world coordinate system or to each camera, together with the associated camera calibration parameters. In order for the comparison to be meaningful, we have modelled a virtual replica of the *Ballet* scene, with reference cameras placed exactly at the same place as the outermost cameras of the original dataset.

Figure 3.16 shows the results of reprojecting the depth of one reference camera onto a virtual camera. The global Z_w axis points up, and the quantization is coarser the farther from the floor. It clearly shows that, even for the finer steps of quantization, the background is visible through the main characters and, for the coarser levels of quantization, the shape that is being described is quite removed from what it should be.

Of course, when the contributions from the other camera are added into the mix, the holes between the slices are filled by whatever is visible through them, resulting in baffling results even though there is no noise, proving the point that using both global depth and depth quantization is really not a good idea. Moreover, as the angles between visual rays from the reference cameras and the global Z axis approach 90° , the coded surface is completely corrupted despite the fact that we had true depth data to start with, and just by using a wrong convention we managed to destroy it.

Whilst this camera configuration could seem far-fetched, it is obvious that in a system aiming to offer 360° navigation some of the cameras would necessarily be off-axis in a similar way.

Figure 3.17 shows a final render of the same scene, now using local depth maps. Even though they are still quantized to 8-bit values, they perform much better, achieving excellent

Figure 3.17: Final depth estimation and final results of backward mapping using local reference depth data.

reconstruction of the main characters.

We have therefore shown that coupling global reference with quantization is a bad choice for depth representation. However, even in the absence of quantization, global reference is still a bad choice because some points may not be uniquely defined. Let us first recall equation (3.6), which describes the unprojection of a point with known camera-local Z coordinate Z_c :

$$\tilde{\mathbf{X}}_c = sK^{-1}\mathbf{x},$$

where s is a scalar chosen so that the last element of $\tilde{\mathbf{X}}_c$ equals the known Z_c ; $\mathbf{x}$ represents a regular point (i.e., not at infinity) on the image plane or, equivalently, $(0, 0, 1) \cdot \mathbf{x} \neq 0$; since the inverse projection matrix

$$K^{-1} = \begin{pmatrix} \frac{1}{\alpha_x} & -\frac{x_0}{\alpha_x} \\ \frac{1}{\alpha_y} & -\frac{y_0}{\alpha_y} \\ 1 & 1 \end{pmatrix},$$

it is immediate that $(0, 0, 1) \cdot K^{-1}\mathbf{x} \neq 0$; therefore s is unique and the 3D point $\tilde{\mathbf{X}}_c$ is well defined. However, when the depth is encoded in terms of a world Z coordinate Z_w , equation (3.7) applies instead:

$$\tilde{\mathbf{X}}_w = sR^\top K^{-1}\mathbf{x} + \tilde{\mathbf{C}},$$

where s is a scalar chosen so that the last element of $\tilde{\mathbf{X}}_w$ equals the known Z_w . The key difference, again assuming that $\mathbf{x}$ is a regular point, is that it is entirely possible that $(0, 0, 1) \cdot R^\top K^{-1}\mathbf{x} = 0$, which geometrically means that the ray passing through $\tilde{\mathbf{C}}$ and $\mathbf{x}$ is contained within a plane with constant world- Z . In this case, it is obvious that every s will be valid if $Z_w - (0, 0, 1) \cdot \tilde{\mathbf{C}} = 0$ and no s will satisfy the conditions if $Z_w - (0, 0, 1) \cdot \tilde{\mathbf{C}} \neq 0$, though the latter case can never happen if $\mathbf{x}$ originated from a real point.

Therefore, our recommendation is to always use local depth maps regardless of the quantization/range compression scheme that may be used to encode depth images.

3.4 Implementation notes

Every step in the described DIBR algorithm has been implemented in a GPU because it lends naturally to a data-parallel description and therefore is a prime candidate for execution in such device. Actually, all the numerous matrix multiplications that take place to unproject points and transform them into other reference systems are indeed the same kind of operations that GPUs execute during the vertex transformation and lighting phase of a classical computer graphics pipeline, so the GPU excels at them and has allowed us to implement this algorithm at real-time frame rates (this system was originally implemented on an NVIDIA GT 220 GPU using CUDA and it did not quite reach real-time performance but, since the algorithm is data-parallel, more recent GPUs are perfectly capable of real-time execution).

The only stage where something not strictly parallel may happen is when obtaining the new virtual depth maps for the virtual camera. The 2D pixels from the reference camera have been already transformed into 3D points in the reference system of the virtual camera, but since perspective has changed, many points may project onto the same pixel, and the closest one to the camera should determine the value of the depth map. There are (at least) two ways to map data to execution threads:

1. Map execution threads to pixels of the virtual depth map. Each thread will iterate over all the 3D points of the point cloud to determine whether the current value of its assigned pixel in the depth map should change. This approach is completely data-parallel but terribly inefficient because most of the 3D points will be irrelevant to the current pixel.
2. Map execution threads to pixels of the reference depth map or, equivalently, 3D points. Each thread will compute the pixel of the new virtual depth map where the point should project and write its camera-local Z value on it. This approach means considerably less per-thread work but the contents of the virtual depth map will be different depending on the thread scheduling order because many concurrent threads may attempt to modify the same pixel.

Fortunately, modern GPUs (starting with CUDA 1.1-capable devices from 2008 onwards) provide atomic compare-and-swap (CAS) instructions that allow to test the value of a memory address and conditionally change it in a single, indivisible operation, which helps ensure correctness regardless of the thread scheduling order. As explained in section 2.4.2, all threads in the same warp execute synchronously. However, it is obvious that such atomic instruction cannot be synchronously executed on the same memory address by all the threads of a warp; they must be serialized just as divergent branches (see figure 2.12), thereby reducing throughput. However, we have found that this throughput reduction only occurs when collisions are indeed bound to happen within the same warp, but not otherwise. Therefore, we have implemented this step of the algorithm using the latter of the two options.

3.5 Conclusions

We have explored a wide baseline, convergent camera configuration using DIBR techniques to assess their suitability for reconstructing the main characters in a scene and allowing the user to choose a virtual viewpoint with more freedom than the usual short baseline, parallel camera configuration usual in 3D TV affords, which is one of the usual applications where DIBR has been used traditionally.

In order to carry out this study, we have implemented our own backward mapping view interpolation algorithm, featuring a simple depth inpainting scheme based on median filters, on

a GPU. This has allowed us to visualize and interact with the scene in real time due to the excellent match between DIBR techniques and the parallel architecture of GPUs.

We have shown, using a synthetic version of one of our original scenes, that backward mapping DIBR achieves excellent quality in the reconstruction of the main characters of the scene even using only two cameras as long as high quality depth information is available. Moreover, we have also shown that, even if this information is indeed available, it is necessary to relate it to coordinate systems local to each reference camera in order not to corrupt valuable data.

Chapter 4

Real-time nonparametric background subtraction

Background subtraction, also known as foreground segmentation, is a recurrent task in many computer vision applications which consists in identifying those pixels in each frame of a video stream which correspond to moving objects in the depicted scene (foreground). The results of this task are frequently used as a stepping stone for higher level analysis in applications such as object recognition, tracking, pose estimation or surveillance among others.

The most basic background subtraction strategy one could think of, which actually is what lends its name to the generic task, is just to subtract each input frame from a reference frame which just contains the background of the scene; those pixels with non-zero difference would be classified as foreground. Unsurprisingly, though, such a naive approach does not fare too well with real world scenes, which exhibit a number of typical problems that any robust background subtraction strategy must tackle.

The first problem of this single background reference frame approach, which is especially relevant in surveillance applications, is that a clean reference frame containing only background is necessary; this poses a problem in public spaces, which are constantly being crossed by people or vehicles, making such reference frame somewhat harder to obtain. The reference frame can be captured, even by analog means (see figure 4.1 as an artistic example of a possible background reference frame) just by taking a very long exposure photograph or, equivalently, using a per-pixel central moment estimator, such as the mean or the median, over a long training video sequence. Unfortunately, applications such as surveillance require the system to reach online operation in a short time with very little or no training phase (this constraint is usually known as *bootstrapping*) and therefore this solution is not acceptable. Moreover, if building the background model takes too long, the illumination conditions are likely to change during the acquisition of the reference frame, thereby rendering it useless as a model of the current scene. A robust strategy must be able to cope both with gradual and instantaneous *illumination changes*; therefore, the background model must be continuously updated.

The second fundamental problem with the single background reference frame approach, even if it is updated over time, is that it fails to account for *capture noise*; all video sources contain noise from different sources, both electrical and mechanical. At the very minimum, thermal noise is present in any electronic device, and light sensors in particular also exhibit shot noise due to the discrete nature of light, which is especially relevant in low-light conditions¹. In addition, even

¹Shot noise has a root-mean-square value proportional to the square root of the image intensity, and the noises at different pixels are independent of one another. Therefore, at higher illumination levels it becomes less relevant.

Figure 4.1: *Place de la Concorde*, Paris. This photograph shows a usually very busy intersection devoid of people thanks to an extremely long exposure time. © Lucie de Barbuat and Simon Brodbeck.

cameras whose position is supposed to be fixed (attached to posts, tripods or any other structural element of a building) are not actually immobile: floors, poles and any kind of structure do exhibit vibration (also known as *camera jitter*) induced by passers-by, vehicles or wind. Regardless of the source, electrical or mechanical, the addition of noise to the signal means that it is no longer possible to just classify any pixel with non-zero difference as foreground. The obvious solution is to classify as foreground pixels whose difference with respect to the reference frame is below a certain threshold. However, determining such value requires to strike a good balance: too high will result in low discriminative power and, hence, many misdetections (pixels that should be classified as foreground are classified as background) and too low will result in poor noise filtering and, therefore, many false detections (pixels that should be classified as background are classified as foreground). A robust strategy must be able to adapt to varying noise levels and retain good discriminative capabilities even when the appearance of foreground objects is similar to the background (*camouflage*). In practice, this means that virtually every practical strategy is based on statistical models of the background rather than simple reference frames.

Thus, our initial definition of background subtraction, which amounted to detect those pixels whose value changes over time must be nuanced: it is necessary to detect those pixels whose value changes for reasons other than noise, camera jitter or illumination changes, assuming that the only other reason for value changes are moving objects. In a way, this is mostly true but the results of such a classification would hardly be considered good in many scenarios. It is common for outdoor sequences, for instance, to feature trees, shrubs, flags or water surfaces such as fountains, ponds or lakes. Rarely do those elements remain perfectly still over time, so in a strict sense they are moving objects. However, such classification is not useful and robust strategies must learn to distinguish this *dynamic background*, which is typically characterized by pseudo-periodic appearance changes, from the aperiodic appearance changes that actual moving objects induce. Even harder examples of dynamic background can be found in outdoor sequences under bad weather such as rain, sleet or snow.

Other typical problems found in background subtraction are *shadows* and *reflections* cast by actual moving objects, which substantially modify the appearance of the pixels they occupy

with temporal patterns similar to those of the actual moving objects that cast them. These phenomena are usually too hard to characterise in a general way, so they are often dealt with in later, higher-level stages of a processing pipeline, rather than in the background subtraction stage.

Finally, there is another frequent occurrence that conditions the design of a background subtraction strategy: mobile objects that *briefly* interrupt their motion. In the first place, it is a situation that requires a conscious decision on how to treat it, because there is no universal definition of what “briefly” means. There may be unanimous consensus that a person that crosses the scene, stops for a few seconds and then resumes her motion should clearly be classified as a mobile object. However, situations that are objectively analogous, only varying the time scale, may not elicit the same agreement: how to treat the case of a car that is driven into a parking spot, left there for a few minutes/hours, then driven off again? For some applications, it makes sense to classify it as foreground while for others it must be assimilated to the background over time.

As humans, we do not find any difficulty in these situations. However, we are actually performing object recognition rather than change detection. For background subtraction algorithms, however, this kind of situations poses a problem: if the background model has a fast learning rate, it will perform better in the face of illumination changes, dynamic backgrounds or varying noise levels. Unfortunately, it will also quickly absorb objects when they stop moving. In the end, it is a design choice that will depend on the application at hand and/or will require further processing from higher-level stages to satisfy the specific needs of the final application.

4.1 Related work

Although it might appear a relatively simple problem, background subtraction is an extraordinarily fertile research topic: hundreds of papers have been published over the last two decades using a very wide variety of techniques; many survey and review papers (e.g., [131, 53, 25, 28, 73, 146, 24]) have taken on the daunting task of summarizing and trying to categorize this myriad of existing methods. In this section we will just make a brief review of some of the most popular/successful algorithms and refer the reader to the aforementioned survey papers for a deeper coverage.

There is no universally accepted taxonomy of background subtraction algorithms; some surveys categorize papers in the field according to the type of techniques that they employ, but such classification does not offer the reader significant insight on the expected advantages and weaknesses of each method. Instead, we will employ a functional classification into four roughly orthogonal dichotomies (as used in [35, 99, 28]) which emphasize the general traits of methods belonging to either side of each of them.

Unimodal or multimodal Unimodal methods are those, regardless of the specific technique they use, that derive from the assumption that the appearance values of a pixel follow a unimodal probability distribution. This usually translates into simpler and faster designs that may perform well enough in many indoor scenarios but that fail catastrophically in sequences exhibiting heavy noise, sudden illumination changes or dynamic backgrounds; possibly the two most represented groups of unimodal strategies in the literature are:

- **Temporal Filters:** The basic idea of these algorithms is to store the latest n values for each pixel and apply a temporal filter, which can be simply a running average [92] or a costlier yet more robust estimator like the median [107, 97, 41], to obtain *the* correct value for the background. Temporal filtering by itself is a poor method for many real-world problems, even in indoor sequences, and most of these systems need to apply further

processing steps to try to correct its deficiencies. Furthermore, the absence of statistical measures of the sampled values impairs automation.

- **Single Gaussian:** One of the simplest possible statistical models is to assume that the values of each pixel conform to a Gaussian distribution, whose mean and variance are continuously updated as new images arrive [164, 86], classifying outliers as foreground. While these methods are clearly inadequate for complex outdoor scenarios, they are very fast and remain competitive [146] in controlled environments.

Multimodal methods, conversely, are those that do not assume that the appearance values of a pixel follow a unimodal probability distribution and, therefore, use a model that can recognize clearly different pixel appearances as belonging to the background. Such models are obviously more complex than the aforementioned unimodal proposals and consequently carry a higher computational cost, but their increased robustness makes them more desirable in generic scenarios. Hence, the vast majority of recent proposals in the literature (and all the rest in this section) are multimodal.

Parametric or nonparametric Parametric models are those that assume that the probability distribution function of the appearance values of a pixel can be described by a specific family of distributions; the specific distribution from the family that is used at a particular moment depends on a number of parameters. One of the simplest possible examples would be the aforementioned single Gaussian model [164], whose parameters would be the mean and the variance of the Gaussian distribution that models the variation of the pixels. More generally, the design of parametric models reflects assumptions on the distribution of the input data and tries to adjust observations to a preconceived model; as long as the input data conform well with these assumptions, results can be excellent. However, when the input data violate these assumptions, the performance of parametric models decreases drastically; in addition, they tend to require careful tuning of their parameters for each different scenario, which decreases their usability.

Some of the most successful families of parametric strategies are:

- **Gaussian mixture models (GMM):** In order to overcome the intrinsic weakness of the single Gaussian approach when faced with dynamic backgrounds, the next logical step is to use several Gaussians [148] to model the distribution of pixel values. This strategy has proved immensely popular and has been used as a starting point for hundreds of researchers [25] due to its relatively low complexity and good results in many typical scenarios. While it constitutes a significant improvement over single Gaussian models, each scenario typically requires a different number of modes and associated parameters for optimal operation, which makes it cumbersome to use. Consequently, enhancements have been proposed to try to automatically estimate the necessary number of modes and reduce the number of parameters to be adjusted [170, 70]. Even with these improvements, highly complex dynamic backgrounds still pose a challenge for GMM-based methods.
- **Hidden Markov models (HMM):** Strategies based on HMM [135, 18, 160] try to cope with dynamic backgrounds by introducing an element of temporal coherence in the transition between states (background, foreground, shadow...) to discard too rapid appearance changes as class changing triggers. Due to this temporal coherence they have been used as starting block for action recognition algorithms, but they are not so widely used in general because they require extensive model tuning and training.

Nonparametric models, on the other hand, make no assumptions on the distribution of appearance values of each pixel, but try instead to directly learn it from the observed data. Since there

is no preconceived model to adjust to the actual data, their behaviour is dictated by the data themselves, which makes it difficult to predict their performance in some scenarios. However, nonparametric models generally tend to be more flexible and require considerably less tuning, which increases automation.

Some of the most popular families of nonparametric strategies are:

- **Kernel density estimation (KDE):** Instead of fitting a model to the samples as GMMs do, the basic idea [51] is to convolve with a kernel function the most recent values of each pixel for directly estimating its value distribution. The foreground distribution is modelled in the same way using as reference data only those past samples that were classified as foreground. The only parameter that such a model needs is the width (usually referred to as bandwidth in KDE algorithms) of the kernel function, but there are proposals to automatically estimate this as well [51, 23, 80]. Further additions are the use of combined space-appearance kernels [142], selective update mechanisms [9] and dynamic per-pixel threshold setting in the Bayesian foreground-background classifier [78]. KDE-methods perform very well with dynamic backgrounds but constructing the density estimation is compute- and memory-intensive.
- **Artificial neural networks (ANN):** One of the most promising families of methods are detectors based on unsupervised neural networks that learn background patterns from the input data. Different types of networks have been described in the literature: general regression [45], competitive [98] or self-organizing [99, 100]. Some of these methods perform very well if properly trained, but present some problems in bootstrapping situations where there is no available clean background to train the network.

Recursive or nonrecursive Nonrecursive techniques (most KDE [51, 142, 141, 9, 78] and temporal filter [107, 97, 92] approaches, among others) construct a “new” background model for each input frame, based on a buffer of recent samples for each pixel. Recursive techniques, on the contrary, do not explicitly store past history of the pixels but instead update a single model based on each new input frame. Nonrecursive techniques tend to have better adaptability than recursive methods because their memory of past events is limited to the size of the buffer, while recursive models can be influenced by events in the distant past; in particular, classification errors can have a negative influence on the model for a long time.

Logically, the tradeoffs involved are that, since nonrecursive methods construct their model on past samples, it is necessary to store many past samples and they are generally more compute intensive as well, while recursive methods require less memory.

Pixel- or region-based Pixel-based methods (e.g. [164, 148, 135, 50, 97, 170, 99]) model each pixel independently of its neighbours, while region-based methods (e.g. [51, 142, 100]) take into account spatial coherence of images to build more robust models in which the model of each pixel also considers its neighbours. Obviously, region-based models tend to be more complex and compute-intensive, so many methods take a hybrid approach, using a pixel-based model followed by a region-based regularization step (e.g. [148, 135, 127, 70]), which is typically less costly than full region-based modelling.

4.2 System overview

In order to achieve a long-term stable algorithm, which is important in applications that must work continuously (e.g., surveillance), nonrecursive methods should be preferable to recursive

Figure 4.2: Block diagram of the proposed system. Notation: round-edged blocks denote data and rectangular blocks denote processes. The green and orange blocks indicate, respectively, the input and the output of the system.

ones. However, most methods in the literature are recursive because these are considerably less computationally expensive. Among nonrecursive methods, KDE-based methods have shown promise, but they have not been very widely used because their memory and processor requirements are huge. However, the emergence of GPUs has enabled us to further pursue research on KDE methods while achieving real-time performance.

In this chapter, we present a nonparametric foreground segmentation system based on the KDE paradigm, augmented with a particle-filter-based tracking stage for updating the foreground model, as well as its real-time implementation on a CPU-GPU heterogeneous system. This proposal can be categorised, following the taxonomy of section §4.1, as multimodal, nonparametric, nonrecursive and region-based.

The main blocks of the proposed detection system and their interconnections are shown in figure 4.2.

For each frame in an input sequence, the probability density functions (PDFs) that each image pixel belongs to the image background or foreground are nonparametrically estimated using spatio-temporal Gaussian kernels as described, respectively, in sections 4.3.1 and 4.3.2. These models are introduced in a novel Bayesian classifier that allows to combine PDFs obtained with spatio-temporal kernels of different spatial bandwidths. The result provided by this classifier, detailed in section 4.3.3, is a binary mask determining what pixels belong to the moving objects in the image.

Then, this mask is introduced in a tracking module. In this module, described in section 4.4, a new tracking strategy based on the simultaneous application of multiple particle filters is used, which allows to deal with an unknown and variable number of moving regions while maintaining an approximately constant computational cost. The data resulting from the region tracking is used to update the spatial coordinates of the reference foreground data, which improves the quality of the foreground modeling and allows using kernels with small spatial bandwidth. Additionally, the filters provide prior information that is fed back into the Bayesian classifier to enhance discrimination between foreground and background.

Finally, taking into account the binary mask obtained for the current image and the tracking results, a region of interest estimation is performed which, as described in section 4.5, results in a drastic reduction the computational cost in the estimation of both background and foreground models.

Figure 4.3: Estimation of a density function (ground truth in grey) based on samples (blue) with different bandwidths: too small (red), too large (green) and correct (black). Figure courtesy of Tarn Duong.

4.3 Spatio-temporal nonparametric modelling

The generic technique of KDE was proposed several decades ago [137, 129]. The basic idea is that, given a sufficiently large number of independent samples $(x_1, x_2, \dots, x_N)$ of a random variable with unknown pdf f , this unknown density can be approximated as

$$\hat{f}_\sigma(x) = \frac{1}{N} \sum_{i=1}^N K_\sigma(x - x_i), \quad (4.1)$$

where $K_\sigma(x) = \frac{1}{\sigma} K(\frac{x}{\sigma})$; K is known as a *kernel* function that should be a probability density function itself (i.e., be non-negative and integrate to one), be unimodal and have zero mean, and σ is a scaling/smoothing factor commonly known as bandwidth or simply width. Many kernel functions have been proposed in the literature [162], each with its own associated trade-offs, but arguably one of the most widely used kernels is the standard normal distribution because of its many desirable mathematical properties.

The adequate selection of the bandwidth σ is critical (see figure 4.3) to obtain a good approximation of f ; as a general rule, the denser the samples, the smaller σ can be, but in practice no single algorithm seems to be best for all problems, hence many bandwidth selection algorithms have been proposed in the literature [83]. In fact, since the local density of samples typically exhibits wide variations, it may be desirable to use variable bandwidths across the domain of f , either depending on the location of the test point (*balloon* estimation) or on the location of each sample (*pointwise* or *sample-point* estimation).

4.3.1 Background modeling

The idea of KDE methods can be adapted to moving object detection to model the probability distribution of values for each pixel. In this work, we start from the joint spatio-temporal model

proposed in [142] and extend it with a selective update mechanism and two proposals for dynamic bandwidth estimation.

Let p^n be a pixel in the current image I^n at time n . Let such pixel be defined by a $(D+2)$ -dimensional vector, $\mathbf{x}^n = ((\mathbf{a}^n)^\top, (\mathbf{s}^n)^\top)^\top$, where $\mathbf{a}^n$ is a D -dimensional vector containing appearance information (e.g. color or gradient magnitude) of the pixel and $\mathbf{s}^n = (h^n, w^n)^\top$ is a vector containing its spatial coordinates (row and column). Let $\{\mathbf{x}_\beta^i\}_{i=1}^{N_\beta}$ be a set of N_β $(D+2)$ -dimensional reference samples, obtained from T_β previous images into a spatial neighborhood around (h^n, w^n) . Applying Gaussian kernels with diagonal covariance matrices², $\Sigma_{\beta, \mathbf{x}^n} = \text{diag}(\sigma_{\beta,1}^2, \sigma_{\beta,2}^2 \dots \sigma_{\beta,D}^2, \sigma_{\beta,H}^2, \sigma_{\beta,W}^2)$, the pdf that p^n belongs to the image background, β , is estimated as

$$p(\mathbf{x}^n|\beta) = \frac{1}{\sum_{i=1}^{N_\beta} w_i} \cdot \frac{1}{(2\pi)^{1+\frac{D}{2}}} \cdot \frac{1}{|\Sigma_{\beta, \mathbf{x}^n}|^{\frac{1}{2}}} \cdot \sum_{i=1}^{N_\beta} w_i \prod_{j=1}^{D+2} \exp\left(-\frac{(\mathbf{x}^n(j) - \mathbf{x}_\beta^i(j))^2}{2\Sigma_{\beta, \mathbf{x}^n}(j, j)}\right) \quad (4.2)$$

where w_i is the weight assigned to the i -th reference sample (in a basic model with blind update, this weight is always one).

To prevent the evaluation of data not contributing significantly to this estimation, the spatial neighborhood is limited to the samples satisfying $D_{Mah}(\mathbf{s}^n, \mathbf{s}_\beta^i) \leq d_{\max}$, where $D_{Mah}(\mathbf{s}^n, \mathbf{s}_\beta^i)$ is the Mahalanobis distance between the spatial coordinates of the current sample and the i -th reference sample using $\Sigma_{\beta, \mathbf{s}^n} = \text{diag}(\sigma_{\beta,H}^2, \sigma_{\beta,W}^2)$ as covariance matrix, and $d_{\max} = 3$. Thus, we only consider those reference samples falling inside the central region that accounts for 99% of the probability of the spatial Gaussian kernels defined by $\sigma_{\beta,H}^2$ and $\sigma_{\beta,W}^2$ [62].

4.3.1.1 Dynamic bandwidth selection

As is the case for any KDE-based application, correct bandwidth selection is paramount for nonparametric background modelling: too high a bandwidth will result in misdetections and too low will result in false detections. Some authors resort to manually setting it for each scenario [142, 168] to save computational resources and achieve higher processing speed, while others propose algorithms to automatically estimate the best bandwidth [51, 103, 154], logically at the cost of more processing time.

In our case, as the background reference samples are uniformly distributed in space, we use fixed bandwidths for the spatial components on the whole image extent. However, to achieve the best balance between misdetections and false detections, we estimate dynamically adequate bandwidth values for the appearance components in each pixel. Then, all the pixel-wise estimations in the neighbourhood of each pixel are weighted and averaged to yield the final bandwidths as

$$\sigma_{\beta,j} = \frac{\sum_{h,w} w_\Sigma(h, w) \Sigma_{\beta,h,w}(j)}{\sum_{h,w} w_\Sigma(h, w)} : j \in [1, D], \quad (4.3)$$

where $\Sigma_{\beta,h,w}$ is the initial pixel-wise bandwidth obtained from the set of differences between consecutive reference samples at the coordinates (h, w) , $\{\Delta \mathbf{a}_{\beta,h,w}^i\}_{k=1}^{T_\beta}$, and $w_\Sigma(h, w)$ is a weight

²This means that each of the components of $\mathbf{x}^n$ is treated as independent; while this may not be strictly true, using completely parameterized covariance matrices [113, 154] results in dramatically increased cost. Assuming these components are independent [51, 142, 103] is a reasonable compromise between quality and efficiency.

assigned to each subset of differences, according to its spatial distance to p^n . As the influence of the reference samples over each current pixel decreases with their spatial distance to that pixel, these weights are obtained through the evaluation of spatial Gaussians at these distances:

$$w_\Sigma(h, w) \propto \exp\left(-\frac{(h^n - h)^2}{2\sigma_{\beta, H}^2} - \frac{(w^n - w)^2}{2\sigma_{\beta, W}^2}\right). \quad (4.4)$$

These Gaussians are computed simultaneously with $p(\mathbf{x}^n|\beta)$. Therefore, their use in equation (4.3) does not involve any increase of the computational cost.

$\Sigma_{\beta, h, w}$ can be estimated using the medians of the absolute value of these differences [51]. However, medians are fairly expensive to compute (between $O(\log T_\beta)$ and $O(T_\beta)$ depending on what data structures can be used: in a GPU, for example, tree structures are not practical), so we propose two alternative methods for computing this initial bandwidth.

Weighted mean. Instead of the median, we can use a weighted running average as a less costly ($O(1)$) estimator,

$$\Sigma_{\beta, h, w}(d) = \sqrt{\frac{\sum_{i=1}^{T_\beta} \Pr(\beta|\mathbf{x}_{\beta, h, w}^i) \left(\Delta \mathbf{a}_{\beta, h, w}^i(d)\right)^2}{2 \sum_{i=1}^{T_\beta} \Pr(\beta|\mathbf{x}_{\beta, h, w}^i)}} : d \in [1, D], \quad (4.5)$$

where $\Pr(\beta|\mathbf{x}_\beta^i) = 1 - \Pr(\phi|\mathbf{x}_\beta^i)$ is the probability assigned to the reference sample $\mathbf{x}_\beta^i$ of being part of the background. Weighing the differences by these probabilities, the negative influence of outliers is reduced. Moreover, as the probabilities of the reference samples to belong to the background have been previously obtained, they can be used without any additional computational effort.

Unfortunately, this estimation method depends on past classifications to work well, and therefore misdetections tend to upset the estimation and, hence, further detections; while its cost is very low, it tends to work less reliably under bootstrapping constraints, as shown in figure 4.4.b.

Bandwidth switching. From extensive testing, we have observed that the results yielded by different fixed bandwidths do not vary significantly as long as these bandwidths are of similar *magnitude*; therefore, all we need to estimate is the correct order of magnitude of each appearance bandwidth component. Thus, we propose a robust and efficient estimator based on per-channel log-spaced histograms.

Absolute differences between consecutive samples with normalized data fall, by definition, into the range $[0, 1]$, so we partition this interval in bins defined by the knots

$$\{0\} \cup \left\{ \exp\left(\log(\sigma_{\min}) \left(1 - \frac{i-1}{N_\sigma - 1}\right)\right) \right\}_{i=1}^{N_\sigma}, \quad (4.6)$$

where $\sigma_{\min}$ is the minimum meaningful difference in the appearance space that is being used and N_σ is the number of bins of the histogram. In each frame, one bin is incremented and another decremented (the ones corresponding to the newest difference and the oldest difference used in the previous frame, respectively) with cost $O(1)$, then the cumulative sum is computed ($O(N_\sigma)$, independent of T_β) to find the bin b_d where the true median is contained. The greater N_σ , the more this estimator converges to the median. Finally, the bandwidth is computed as

Figure 4.4: Detections only using the background model in an outdoor sequence using different appearance width estimators: (a) Original image; (b) detection using the weighted mean; (c) detection using the median; (d) detection using the proposed bandwidth switching scheme. Colour notation (for this figure and all subsequent ones showing detection results): correct detections (green), misdetections (red), and false detections (black).

$$\Sigma_{\beta,h,w}(d) \stackrel{[51]}{=} \frac{m_d}{0.68\sqrt{2}}, \quad (4.7)$$

$$m_d = \begin{cases} \sigma_{min} & b_d = 1 \\ \exp\left(\log(\sigma_{min}) \left(1 - \frac{2b_d-3}{2(N_\sigma-1)}\right)\right) & b_d \in \{2, \dots, N_\sigma\} \end{cases}.$$

As we discuss in section §4.7, even for low N_σ the obtained results are virtually undistinguishable from using the median (compare (c) and (d) in figure 4.4).

4.3.1.2 Selective update

The nonrecursive nature of the proposed nonparametric model makes it very quick to adapt to changes in the background. However, this may pose a problem when foreground objects do not move fast enough because the background model is just as quick to learn their appearance and absorb them if all past pixels are blindly added to the pool of background reference samples. As exposed in the introduction of this chapter, this may or may not be desirable, depending on the intended application of the proposed system.

A possible solution to increase the permanence of the foreground is using pure selective update, which consists in only adding to the background reference sample pool those pixels that are not classified as foreground (or equivalently setting the weight of samples classified as foreground to zero in equation (4.2)). This effectively prevents foreground pixels to be used to model the background and causes foreground objects that stop their motion to be detected forever. Moreover, false detections also become permanent, which is the main reason why this approach is rarely used.

Some authors try to control the absorption rate by simultaneously applying several models with different time frames/learning rates [52, 132, 55] (i.e. combining a short-term model and a long-term model); logically, the computational cost of such approaches is also higher.

Figure 4.5: Temporal width used to perform the selective update of the background models.

We propose a gradual selective update mechanism that allows the user to easily control the tradeoff between permanence and adaptability by simply adjusting a temporal window: samples from the recent past are treated in a probabilistic selective way, while samples from the more distant past are introduced into the model in an increasingly blind manner. Thus, the weights w_i in equation (4.2) are set as

$$w_i = 1 - \vartheta \Pr(\phi | \mathbf{x}_\beta^i), \quad (4.8)$$

where $\Pr(\phi | \mathbf{x}_\beta^i)$ is the probability assigned to the reference sample $\mathbf{x}_\beta^i$ of being part of the foreground (see section 4.3.3) and ϑ is a temporal weight obtained as

$$\vartheta = \begin{cases} 1 & \text{if } \Delta n \leq T_0 \\ G_T(\Delta n) & \text{if } \Delta n > T_0 \end{cases}. \quad (4.9)$$

In this temporal weight, illustrated in figure 4.5, Δn is the temporal distance between the reference samples and the current one, $T_0 < T_\beta$ is a predefined constant value that controls the absorption rate of the model and $G_T(\Delta n)$ is a temporal Gaussian defined as

$$G_T(\Delta n) = \exp\left(-\frac{(\Delta n - T_0)^2}{2\sigma_T^2}\right), \quad (4.10)$$

where $\sigma_T = (T_\beta - T_0)/3$ so that $G_T(T_\beta) \approx 0$.

Thus, on the one hand, the contribution of the T_0 most recent background reference samples only depends on their probability to be classified as background, $\Pr(\beta | \mathbf{x}_\beta^i) = 1 - \Pr(\phi | \mathbf{x}_\beta^i)$. Therefore, assuming that, at a given time, the moving objects are correctly detected, the samples of such objects will just barely affect the background model corresponding to the following T_0 images. Consequently, even if the moving objects stop moving, during that period their absorption by the background model will be negligible (the absorption could vary slightly depending on the exact probability values assigned to the reference samples).

On the other hand, the probability values associated with the reference samples with $\Delta n > T_0$ lose relevance in the modeling as the value of Δn increases. Therefore, all the reference samples (whether they have been classified as foreground or background) will end up influencing the model. In this way, moving objects remaining static will become part of the background and false detections will disappear.

4.3.2 Foreground modeling

The pdf that p^n belongs to the image foreground, ϕ , can be estimated [142] as a mixture of a constant density, γ , of a uniform random variable in the $D+2$ components defined for the feature

vector $\mathbf{x}^n$ and a density function estimated with Gaussian kernels:

$$p(\mathbf{x}^n|\phi) = \alpha\gamma + \frac{1-\alpha}{N_\phi (2\pi)^{1+\frac{D}{2}}} \cdot \frac{1}{|\Sigma_{\phi, \mathbf{x}^n}|^{\frac{1}{2}}} \cdot \sum_{i=1}^{N_\phi} \prod_{j=1}^{D+2} \exp\left(-\frac{(\mathbf{x}^n(j) - \mathbf{z}_\phi^i(j))^2}{2\Sigma_{\phi, \mathbf{x}^n}(j, j)}\right) \quad (4.11)$$

where α is a mixture factor, $\Sigma_{\phi, \mathbf{x}^n} = \text{diag}(\sigma_{\phi,1}^2, \sigma_{\phi,2}^2 \dots \sigma_{\phi,D}^2, \sigma_{\phi,H}^2, \sigma_{\phi,W}^2)$ is the covariance matrix used in the kernels and $\{\mathbf{z}_\phi^i\}_{i=1}^{N_\phi}$ is the set of foreground samples stored along the last T_ϕ images into a spatial neighborhood around the coordinates of p^n . Just as in the background modeling, to avoid the evaluation of reference samples not contributing significantly to the estimation process, only samples satisfying

$$D_{Mah}(\mathbf{s}^n, \mathbf{s}_\phi^i) \leq d_{max} = 3, \quad (4.12)$$

with covariance matrix $\Sigma_{\phi, \mathbf{s}^n} = \text{diag}(\sigma_{\phi,H}^2, \sigma_{\phi,W}^2)$, are considered.

The spatial width values used in the foreground model depend on the number of reference images, T_ϕ , and on the speed of the moving objects: they should be large enough to take into account all the reference data in all the reference images [42]. Therefore, in principle, the spatial widths used in the foreground should be significantly larger than those used in the background [142], which results in a high computational cost. As discussed in section 4.4, the application of the proposed tracking strategy allows to reduce these widths and make them independent of the speed of the objects and the amount of reference images.

Appearance bandwidths cannot be determined using the same procedure of the background because the distribution of reference data is not dense or regular enough. In addition, variance is not only caused by capture noise but also by the changes in the observer's angle of view. Therefore, we resort to empirically setting this parameter, which is relatively insensitive in most sequences. The most important phenomenon that must be noted is that if the foreground widths are set too small, in the same order of magnitude as those used in the background, false detections tend to feed back and become persistent due to the combined effect of the selective update of the background model and the increased prior probability of the foreground in the classifier (see section 4.3.3). Therefore, it is recommendable to use larger values than those estimated for the background.

4.3.3 Conditioned Bayesian classifier

As said before, on the one hand, the spatial width used in the background modeling must be small. However, on the other hand, the width applied to the foreground modeling must be large enough to cover the object displacements along the reference images. Consequently, it is desirable to use different width values in each model. However, foreground and background reference data are distributed in space very differently, which may result in significant mismatch in the magnitudes of $p(\mathbf{s}^n|\phi)$ and $p(\mathbf{s}^n|\beta)$ in some regions, leading to persistent false detections. Therefore, instead of the typical Bayesian classifier [103], we propose an alternative to decouple the appearance and spatial information of both models:

$$\Pr(\phi|\mathbf{x}^n) = \frac{\Pr(\phi|\mathbf{s}^n) p(\mathbf{x}^n|\phi, \mathbf{s}^n)}{\Pr(\phi|\mathbf{s}^n) p(\mathbf{x}^n|\phi, \mathbf{s}^n) + \Pr(\beta|\mathbf{s}^n) p(\mathbf{x}^n|\beta, \mathbf{s}^n)} \quad (4.13)$$

where $\Pr(\phi|\mathbf{s}^n)$ and $\Pr(\beta|\mathbf{s}^n) = 1 - \Pr(\phi|\mathbf{s}^n)$ are, respectively, the foreground and background prior probabilities (obtained from the results provided by the tracking strategy as described in section 4.4); and $p(\mathbf{x}^n|\phi, h^n, w^n)$ and $p(\mathbf{x}^n|\beta, h^n, w^n)$ result from conditioning the foreground and

Figure 4.6: (a) Original images from two video sequences. (b) Detections using the typical Bayesian classifier. (c) Detections using the proposed conditioned classifier.

background models, $p(\mathbf{x}^n|\phi)$ and $p(\mathbf{x}^n|\beta)$, on a particular spatial location. These conditioned density functions are obtained as

$$p(\mathbf{x}^n|\xi, h^n, w^n) = \frac{p(\mathbf{x}^n|\xi)}{p(h^n, w^n|\xi)} : \xi \in \{\beta, \phi\} \quad (4.14)$$

where $p(h^n, w^n|\xi)$ is the marginalization of $p(\mathbf{x}^n|\xi)$ over the D -dimensional set of appearance characteristics. These marginal density functions are obtained as

$$p(h^n, w^n|\beta) = \frac{1}{\sum_{i=1}^{N_\beta} w_i} \cdot \frac{1}{2\pi} \cdot \frac{1}{\sigma_{\beta,H}\sigma_{\beta,W}} \cdot \sum_{i=1}^{N_\beta} w_i \prod_{j=D+1}^{D+2} \exp\left(-\frac{(\mathbf{x}^n(j) - \mathbf{x}_\beta^i(j))^2}{2\Sigma_{\beta,\mathbf{x}^n}(j,j)}\right) \quad (4.15)$$

$$p(h^n, w^n|\phi) = \alpha\gamma' + \frac{1-\alpha}{N_\phi 2\pi} \cdot \frac{1}{\sigma_{\phi,H}\sigma_{\phi,W}} \cdot \sum_{i=1}^{N_\phi} \prod_{j=D+1}^{D+2} \exp\left(-\frac{(\mathbf{x}^n(j) - \mathbf{z}_\phi^i(j))^2}{2\Sigma_{\phi,\mathbf{x}^n}(j,j)}\right) \quad (4.16)$$

where γ' is a constant density in the spatial components. Note that these marginal densities can be obtained without any extra computational effort (compare equations (4.15) and (4.16) to equations (4.2) and (4.11)). Figure 4.6 illustrates some results obtained with the typical Bayesian classifier and with the proposed classifier. Although moving objects are correctly detected in both cases, the typical classifier yields many persistent false detections, whereas the proposed one mitigates them.

4.4 Object tracking with a particle filter

The model of the foreground uses as reference data only those pixels that were classified as foreground in past frames instead of the whole images [142]. However, the foreground is made up, by definition, of moving objects, and this poses a problem: if we look for reference pixels in

Figure 4.7: Block diagram of the proposed tracking system. Round-edged blocks denote data (green is input; orange is output) and rectangular blocks denote processes.

a small spatial vicinity of each current pixel in recent frames, we will find only a few reference pixels unless the object is moving quite slowly; this in turn means that the foreground model will be very noisy and unreliable.

One obvious solution to this problem is increasing the size of the region where reference pixels will be sought. Unfortunately, this is computationally expensive [42] and, moreover, it may result in using pixels from different objects as reference, which decreases the quality of the resulting model. A better solution to the problem is to model the motion of each object in the foreground so that reference data from recent frames can be projected onto the estimated location of each object in the current (to be processed) frame. This both allows to reduce the search area for reference data and increases the probability that the data used to model each pixel is actually relevant to it. Consequently, we propose a multi-object tracking method based on a particle filter framework that is able to deal with an unknown and variable number of moving regions throughout the sequence. Many methods have been proposed for multiple object tracking, but most of them assume object detection has already been performed and work at the object level [111, 145]. Our proposal works instead at a lower level (i.e., connected regions of pixels) and is object-agnostic because its only aim is to compact reference data for the foreground model at a low computational cost.

4.4.1 General description

The proposed algorithm is based on a particle filter of the sequential important resampling (SIR) type [5] and features all its typical stages: prediction, weights update, normalization and resampling. Our goal is to estimate iteratively the probability density function of a state vector $\mathbf{u}^n$ from a set of measures $\mathbf{v}^n$ obtained from the n -th input image, I^n . This probability density function can be defined as

$$p(\mathbf{u}^n | \mathbf{v}^n, \mathbf{v}^{n-1}, \dots, \mathbf{v}^1) \propto p(\mathbf{v}^n | \mathbf{u}^n) p(\mathbf{u}^n | \mathbf{v}^{n-1}, \dots, \mathbf{v}^1), \quad (4.17)$$

where $p(\mathbf{u}^n | \mathbf{v}^{n-1}, \dots, \mathbf{v}^1)$ is the probability distribution predicted from past observations [82] and $p(\mathbf{v}^n | \mathbf{u}^n)$ is the likelihood of the state vector $\mathbf{u}^n$ given the set of J_s measures

$$\mathbf{v}^n = \{\mathbf{v}_m^n\}_{m=1}^{J_s} = \left\{ (h, w) : \Pr(\phi | \mathbf{x}^n) > \frac{1}{2} \right\}, \quad (4.18)$$

which are the locations of the pixels that have been classified as foreground on image I^n . In order to obtain this estimation we evaluate a set of N_s particles $\{\mathbf{u}_i^n, \varpi_i^n\}_{i=1}^{N_s}$, where ϖ_i^n is the weight of the i -th particle in the set.

Figure 4.8: The top row show images from a scene, spaced five frames apart; a gust of wind is blowing the bushes on the left, causing false detections. In the absence of any filtering (middle row), these false detections are fed back into the foreground model, causing persistence. In the bottom row, we show the results of rejecting small regions at the labeling stage. While false detections do occur, they are not fed back into the foreground model, preventing their growth and improving overall detection.

We have chosen to model each moving region as a bidimensional Gaussian distribution whose axes are parallel to the edges of the image, moving with constant velocity from one frame to the next. Thus, each particle will be represented by the state vector defined as

$$\mathbf{u}_i^n = \left(h_i^n, w_i^n, \dot{h}_i^n, \dot{w}_i^n, \sigma_{h,i}^n, \sigma_{w,i}^n \right)^\top, \quad (4.19)$$

where (h_i^n, w_i^n) is the position of the center of the i -th distribution, $(\dot{h}_i^n, \dot{w}_i^n)$ its velocity and $(\sigma_{h,i}^n, \sigma_{w,i}^n)$ its standard deviations in each of its axes.

While particle filters are inherently multimodal, following multiple objects requires significant adaptations of the basic structure of the filter; figure 4.7 shows the block structure of the proposed multi-region tracking algorithm, that comprises stages with global scope as well as stages replicated in a per region basis to better model the independent behaviour of each tracked object. Throughout the following sections, each of the blocks that make up the proposed tracking system are detailed and referred to by their name in the figure.

4.4.2 Multiple object management

For each input image I^n , the N_s particles that the filter uses are distributed among as many groups as contiguous regions are detected in that image according to their area, so that the most significant objects get more precise estimations; since the total number of particles is fixed, the computational cost of the particle filter remains approximately constant. Each particle group receives a unique identifier upon creation and lives for as long as it can be reliably identified with the same moving object over time; for this task, it is useful to represent each group by a meta-particle, which is a weighted sum of all the particles in the group (more details in section 4.4.3).

4.4.2.1 Region identification

The *Label and sort* block implements classic connected-components labeling analysis [149] on the set of measures $\mathbf{v}^n$. Regions too small to be a meaningful object are discarded; this removes most false detections from the pool of reference samples, which avoids undesirable feedback leading to persistence (see figure 4.8). The remaining connected components are measured to obtain their area in pixels, their centre and their standard deviation in the vertical and horizontal axes.

Figure 4.9: Examples of common occurrences that make the number of existing regions change. Red ellipses depict particles and the darker ellipse surrounding objects shows the representative metaparticle of each particle group. In subfigure (a), the person on the left walks behind the table and his foot cannot be connected to the rest of his body; the group of particles assigned to him splits into two groups. In subfigure (b), a man enters the frame, but initially his leg and arm are two independent objects that are later recognized as one, and their associated particle groups joined.

4.4.2.2 Detection of new regions

Among all the regions detected in the *Label and sort* block, some may correspond to objects that were previously detected in the previous frame and some to new objects; we need to determine which (new) regions are not covered by the current set of particles and create new particle groups for them. We consider that a measure is covered by a particle if it is contained within the central region of the distribution associated to the particle that accumulates 99.7% (the common three-sigma rule) of the probability. Therefore, a measure $\mathbf{v}_m^n$ will be covered by the i -th particle $\mathbf{u}_i^n$ if it satisfies

$$\sum_{j=1}^2 \frac{(\mathbf{u}_i^n(j) - \mathbf{v}_m^n(j))^2}{(\mathbf{r}_i^n(j))^2} \leq 1, \quad (4.20)$$

where $\mathbf{r}_i^n \approx 3.44(\sigma_{h,i}^n, \sigma_{w,i}^n)^\top$. The *Compute coverage mask* block performs this computation in the GPU for all possible pixel locations in the image to generate a full binary coverage mask.

The *Determine region coverage* block creates new particle groups for new regions; if none of the measures of a region is covered by the existing particle set, a new particle (group) must be created to start tracking that region. The initialization values will be the metrics of the region as computed by the *Label and sort* block.

4.4.2.3 Disappearance, split and fusion of regions

Now we need to determine the relation between the regions found in the current detection and those found in previous detections to track them correctly. Additionally, we need to take into consideration other common occurrences (see examples in figure 4.9) such as:

- Disappearance of previously existing regions (e.g., a moving object exits the frame or is occluded).

- Fusion of previously disjoint regions (e.g., several parts of an object enter the scene, then the rest of the object connects them; or the projections on the camera of two independent objects overlap; or an object that was partially occluded comes out of the occlusion).
- Split of a previously unique region (e.g., two objects that were together part ways, or an object gets partially occluded, for instance a person passing behind a lamppost).

As this (admittedly non-exhaustive) enumeration shows, a single phenomenon may have different explanations that would require different treatments and a (costly) high-level analysis. However, our proposal is object-agnostic and its only aim is to compact reference data at least not worse than doing nothing. Therefore, we default to treating fused or split regions as new objects; since new particles are created with zero velocity, this is never worse than not using the particle filter.

The *Map regions to particle groups* block implements this policy and yields a bijection between regions in the image and groups of particles. The proposed algorithm is:

- If a particle group (its representative metaparticle) does not cover at all any region, it is immediately deleted.
- We determine which is the preferred particle group for each region (i.e., closest using the Mahalanobis distance for a majority of its measures $\mathbf{v}_m^n$).
- Now we can map each region to its preferred particle group. However, in some cases more than one region can prefer a same group; this indicates that the group is not well adapted to either region. Therefore, we create new groups for all the affected regions.
- Any leftover particle group can simply be deleted.

4.4.2.4 Redistribution of particles

We want to distribute the configured number of particles among the groups/regions we have found so that every group has at least one particle and the number of particles is proportional to the area of their region. These two conditions, in general, are not compatible and we must strike a balance between them. We shall proceed from the smallest to the biggest region, assigning them their fair share or at least one particle. Thus, if a region does not quite reach its fair share, it will necessarily be the biggest and, therefore, have the smallest relative deviation from the share. However, if the configured number of particles is of the same order as the number of regions, this could result in starvation of the biggest regions, so we ensure that, after assigning particles to a region, there must be left at least as many as regions still to be assigned. In this operation, a group of particles can vary with respect to the previous frame. If it needs to be downsized, excess particles are selected at random. If the group gains particles, new particles are created with values randomly distributed around the representative metaparticle of the group.

4.4.3 Region-level tracking

Once all particles have been distributed into groups associated with regions, we perform the typical stages of a particle filter separately on a per particle group/region basis. Thus, particles will be weighed only in relation with the other particles from their group, resampling will take place inside each group and a different dynamic model will be applied to each group.

Firstly, we need to evaluate how well each particle describes the region it has been assigned to. Let us consider a mobile region $V^n = \{V_\tau^n\}_{\tau=1}^J$, which is made of J measures. The weight of the particles that are associated to this region is computed as

$$\varpi_i^n = L(\mathbf{u}_i^n | V^n) \cdot \varpi_i^{n-1}, \quad (4.21)$$

Figure 4.10: Example of two particles with the same F-score on a region with different combinations of precision and recall.

where $L(\mathbf{u}_i^n | V^n)$ is a likelihood function that relates the set of measures belonging to the mobile region to each of the particles associated to that region. We have chosen to evaluate how well a particle adapts to its region in the same terms that are usually employed to measure the performance of classifiers: the harmonic mean of precision and recall, usually known as *F*-score. This likelihood function, illustrated in figure 4.10, does not have any parameters to adjust and produces a well-balanced result.

In this context, we define $J_{\text{in},i}^n$ as the number of measures of the region V^n covered by particle $\mathbf{u}_i^n$. Thus, we can define precision as the fraction of the area covered by a particle (4.20) that contain measures from the region, $P_i^n = \frac{J_{\text{in},i}^n}{\pi \cdot \sigma_{h,i}^n \cdot \sigma_{w,i}^n \cdot 3.44^2}$; we can define recall as the fraction of measures from the region that are covered by a particle, $R_i^n = \frac{J_{\text{in},i}^n}{J}$. Therefore, the *F*-score and likelihood function, is

$$L(\mathbf{u}_i^n | V^n) = \frac{2P_i^n R_i^n}{P_i^n + R_i^n} = \frac{2J_{\text{in},i}^n}{J + \pi \sigma_{h,i}^n \sigma_{w,i}^n 3.44^2}. \quad (4.22)$$

Once all the particles in the group have been weighed, we can update the global state estimation of the group as

$$\mathbf{u}^n = \sum_i \varpi_i^n \mathbf{u}_i^n. \quad (4.23)$$

Weighing all the particles is one of the most computationally demanding stages of the filter and, therefore, we have implemented it in the GPU. Then, we proceed to resample [48] the particles to propagate those with higher weights and eliminate those that did not contribute significantly to the estimation and normalize their weights. Finally, we apply a constant-velocity dynamic model in order to predict their state for the next frame.

4.4.4 Integration of the results with the nonparametric model

The contribution of the particle filter to the main nonparametric model is twofold: it allows us to displace reference data with the objects it belongs to and, since we are estimating the trajectories and velocities of those objects, we can use this knowledge to establish prior probabilities of each pixel to belong to the foreground. In this subsection we will detail both contributions to the nonparametric model.

Figure 4.11: Example of prior foreground probability distribution.

Figure 4.12: Contributions of the tracking system on two scenes. Column (a) shows the original scenes; (b) shows the foreground reference data distribution when the tracking system is not used; (c) shows results without tracking; (d) shows the foreground reference data distribution when its position is displaced according to the dynamic model estimated by the tracking system and (e) shows final detections with clearly improved recall over (c).

4.4.4.1 Prior probability estimation

In the absence of any previous information, the prior probability of a location $\mathbf{s}^n$ in the n -th image to belong to the foreground should be established as $1/2$, which is equivalent to not using prior probabilities. However, after applying the dynamic model to all the previously resampled particles, we expect the particles to be concentrated over the areas where the objects are most likely to be located in the next frame. We can use this information to establish a reasonable prior probability in the areas covered by particles, while retaining the default prior probability where there are no particles in order not to hamper the detection of new objects. Thus, we can define

$$\Pr(\phi|\mathbf{s}^n) = \begin{cases} \frac{1}{2}, & N_p = 0 \\ \frac{1}{2} + \frac{1}{2N_p} \sum_{i=1}^{N_p} G_i(\mathbf{s}^n), & N_p > 0 \end{cases} \quad (4.24)$$

where N_p is the number of predicted particles that cover $\mathbf{s}^n$ (see equation (4.20)) and $G_i(\mathbf{s}^n)$ is the contribution of the i -th particle on that position:

$$G_i(\mathbf{s}^n) = \exp\left(-\frac{(\mathbf{s}^n(1) - h_i^n)^2}{2(\sigma_{h,i}^n)^2} - \frac{(\mathbf{s}^n(2) - w_i^n)^2}{2(\sigma_{w,i}^n)^2}\right). \quad (4.25)$$

This prior is trivially computed on the GPU; figure 4.11 shows a scene and its map of prior probability of the foreground.

4.4.4.2 Displacement of reference data

After all the regions in the current frame have been mapped to particle groups, we can replace the local labels we assigned to the measures $\mathbf{v}^n$ with global labels: the identifiers of their corresponding particle group, which they will, in the absence of fusions or splits, share with measures from previous frames (reference data for the next frame) corresponding to the same object; thus, we can displace all the reference data from the same object using the same dynamic model of its associated particle group; figure 4.12 shows clearly improved reference data densities and results when applying dynamic model to reference data.

4.5 Selective Analysis and Regions of Interest

Arguably the single most significant problem of nonparametric modeling (NPM) consists in its very high computational cost, derived from the fact that to classify each pixel it is necessary to access extensive amounts of data and perform a great number of operations. However, most pixels in typical video sequences belong to the background; therefore, if we can find a way to only analyze regions of interest (RoI) likely to belong to the foreground and classify the rest as background by default we could significantly cut down the computational cost while maintaining the quality of the detections. We propose a GPU-friendly strategy based on three main concepts:

Iterative analysis In most scenarios, relevant moving objects have a minimum size that can be characterized. Therefore, we can divide the input image into blocks smaller than this minimum size and sample them, maintaining a high probability that the moving objects will be at least partially covered by one or more of the sampled blocks and we will detect them by only applying NPM on these blocks. Then, we can iteratively extend NPM analysis only to those blocks next to the ones found to contain foreground, guaranteeing full object detection from a single positive seed block.

Windowed Random Sampling (WRS) We would like to reduce the amount of work as much as possible, and we could do that by sampling the blocks very sparsely. One way to look at it is to divide the image into macroblocks of a few blocks per dimension (e.g. 4×4) and select one block per macroblock to analyze. Unfortunately, this systematic sparse sampling carries the risk of whole objects fitting in the space between samples. However, if the temporal sampling rate of the video is high enough in relation with the speed of the apparent motion of objects in the scene, a moving object will typically spend several frames within a macroblock. Therefore, if instead of selecting fixed blocks within each macroblock we randomly choose one or more blocks to analyze, any moving object will be hit in at most a few frames. Selecting random blocks within macroblocks instead of doing it in the image as a whole ensures that no significant portion of the image can be left out.

Temporal coherency Although the iterative analysis we have proposed guarantees the detection of the full object as soon as one block tests positive, the number of iterations needed for the algorithm to converge may be high. Theoretically, this is not a problem but, in practice, and especially when implementing it in a GPU, it is undesirable for two reasons:

Figure 4.13: Selective analysis of a frame. Subfigure (a) shows an image featuring an already established object and a new object just entering the frame on the left that has not been hit by WRS, hence the misdetection. Subfigure (b) shows, iteration by iteration, the detection process using blocks of 8×8 px, 4×4 macroblocks for WRS and a growing radius of 2-blocks on the very next image (shown in subfigure c), where the new object is correctly detected; blue blocks are those to be analyzed in the next iteration and red/green blocks are those that tested negative/positive. We can see in the zeroth iteration that the blocks where the already established object is expected to be located are marked for analysis from the beginning, and in the first iteration the new object is hit by WRS, correctly growing to full detection in the next iterations. Subfigure (c) shows the actual image and the final detection.

- For the detection of all objects to be guaranteed in every frame, the subsampling should be reasonably dense for all objects to be hit, which means analyzing a significant portion of the image, which is what we want to avoid.
- Results of each iteration of analysis wave must be synchronized and copied to the CPU in order to decide (many GPUs do not allow scheduling new work from the GPU itself) whether a new iteration is needed and which blocks need analyzing this time. Synchronization and data transfer to and from the GPU is costly and should be kept at a minimum, therefore we would like to do as few iterations as possible.

It is, however, possible to reduce the number of necessary iterations just by observing that, very often, the regions occupied by the same moving object in consecutive frames have a significant intersection. Therefore, in addition to the blocks sparsely sampled using WRS, we can choose to analyze in the first iteration all the blocks that contained objects in the previous frame. Thus, we can both reduce the number of necessary iterations for the algorithm to converge and greatly decrease the probability of losing objects. Moreover, since we have already designed a tracking system, the next obvious step is using its predictions and choose to analyze the blocks where we expect to find objects in the next frame.

4.5.1 Detailed algorithm

The detailed implementation of the algorithm, illustrated in figure 4.13, is as follows:

1. Input images are tessellated into blocks. Each block can take one of these states:
 - (a) *not analyzed*: this block has not been analyzed and will not be in the next iteration (this is the initial state of every block). Depicted in white in figure 4.13.

Figure 4.14: Multi-threading in the CPU and asynchronous communication among tasks provide lower end-to-end latency and higher throughput.

- (b) *to be analyzed*: this block will be analyzed in the next iteration of NPM. Depicted in blue in figure 4.13.
 - (c) *positive* or *negative*: this block has been analyzed in a previous iteration of NPM and has or not been found to contain a significant amount of foreground. Depicted in green and red, respectively, in figure 4.13.
2. Whenever a new frame arrives, random blocks are marked *to be analyzed* in accordance with the aforementioned WRS procedure. The state of all other blocks remains unchanged; therefore, if other blocks were already marked to be analyzed, they will be analyzed in addition to the random blocks we have just set.
 3. Blocks marked as *to be analyzed* are so using NPM; then, pixels classified as foreground are counted and the block set to *positive/negative* if the proportion of foreground pixels in the block is greater than a threshold value. If the threshold is high, fewer blocks will be eventually analyzed but the results on the edges of the objects have a higher chance to be wrong; if the threshold is low, the number of blocks that will be analyzed is higher but the quality of the results is improved. Therefore, we have used a very conservative threshold (10%) in our tests.
 4. If there are blocks marked as *positive* that have neighbours marked as *not analyzed*, these neighbours (and optionally more blocks within a certain radius) are changed to *to be analyzed* and we go back again to step 4. Otherwise, the current frame is done. All remaining blocks marked as *not analyzed* are classified as background by default.
 5. All blocks are set to *not analyzed* except those dictated by the temporal coherency criterion: blocks containing a ratio of predicted foreground over the threshold will be set *to be analyzed* in the next frame; we go back to step 2 for the next frame.

4.6 Implementation notes

4.6.1 General architecture

To achieve maximum throughput, the implemented design exploits parallelism not only in the GPU but also in the CPU, offloading input/output operations and their associated computations (decoding and encoding image files) to the host computer. Since input and output operations involve large latencies, separate tasks are created for input, processing control, and output; they communicate and pass data along using circular buffer structures that reside in the global memory

Figure 4.15: Asynchronous memory transfers from and to the GPU allow the processing task to run continuously, thus maximizing throughput.

Figure 4.16: The areas considered for the spatial modeling of the cyan and magenta pixels overlap. The same occurs for all the pixels processed in this block. The data needed for processing the whole tile/block (dark orange) is therefore that depicted in light orange.

of the GPU. This asynchronous coupling among the tasks in the CPU, shown in figure 4.14, absorbs the latencies due to disk reading and writing because input data can be read before the processing thread actually needs it, and output data can be written while the processing thread is already busy with the next images.

This architecture also allows us to take full advantage of the capability of the stream processor that we used to perform asynchronous memory transfers from the CPU to the GPU and back in such a way (see figure 4.15) that memory transfers and processing overlap in time. Thus, the GPU processors do not suffer from data starvation and can be fully utilized the whole time, improving the global throughput.

4.6.2 Nonparametric modelling

The proposed nonparametric models involve a huge number of operations over a very extensive reference data pool. Therefore, to achieve real-time operation, it is crucial to optimise the memory access patterns to achieve the highest possible effective memory throughput, as well as to lower the cost of the operations on the data as much as possible.

4.6.2.1 Memory access

For the spatio-temporal modeling previously defined, we should theoretically consider all data in each reference image for each output pixel. However, since reference pixels are weighted by Gaussian functions, we can restrict our region of interest to the spatial margins (i.e. maximum distance in pixels in each dimension from the pixel under study) $m_{i,j} = \lfloor 3\sigma_{i,j} \rfloor : i \in \{\beta, \phi\}, j \in \{H, W\}$ as explained in section 4.3.1.

We can see in figure 4.16 that each reference pixel is input for many others; input data must necessarily reside in global memory (recall the tiered memory model from section 2.4.3) because it is the only memory zone that is big enough and can be written by the host machine (CPU). In a naïve implementation, each execution thread would directly fetch from global memory the necessary data, thus reading the same data many times over. However, such implementation would achieve very low throughput due to the low effective utilization of the memory bus; it is absolutely necessary to reduce the number of accesses to the global memory. We have evaluated two alternative strategies: cooperative reading and multilayered texture.

Cooperative reading One of the usual patterns for memory access in a GPU arises when the threads in each scheduling block (which in this case are mapped to rectangular tiles in the image) need to read the same data or there is significant intersection between the input of many threads in the block, as is the case; then, a common strategy is to copy all the data needed by all the threads in the block to its shared memory so that each datum is read from the slow global memory only once and all subsequent reads are targeted at the much faster shared memory.

Let us take a closer look at the background model (the case of the foreground model is analogous): the region of interest for a block is the tile mapped to that block, enlarged by $m_{\beta,W}$ pixels at left and right directions and $m_{\beta,H}$ pixels at top and bottom (see figure 4.16). Therefore, each thread block will need to retrieve $2m_{\beta,H} + b_H$ sublines of $2m_{\beta,W} + b_W$ pixels each.

The shared memory region of a block can only be written by the execution threads in that block themselves, which can request data from the global memory and then write them into the shared memory area. The fastest way to perform this memory transfer is to distribute the data among the threads so that they cooperatively read it into the shared memory: each thread requests a different piece from global memory and writes it into the appropriate place on the block's shared memory area. However, recall from section 4.3.1 that some restrictions apply:

- Although an execution thread can request single bytes (the size of each of the channels of a pixel in our images) from global memory, the minimum transfer unit is 32 bytes.
- If the memory addresses of the requests from successive threads in the warp are also successive, the requests can be coalesced into a single memory transaction.
- Transfers from global memory are always of naturally aligned blocks. This is, if there are two threads in a warp whose requests are for bytes with different addresses modulo 32, there will be necessarily two separate memory transactions³, even if all the addresses in the warp are successive.

Let us assume, for the sake of clarity, that the first byte in every line is 32-byte aligned. As figure 4.17 shows, we need to read pixels in the horizontal range $[h_0 - m_{\beta,W}, h_0 + b_W + m_{\beta,W} - 1]$, h_0 being the horizontal coordinate of the leftmost pixel in a given tile/thread block, or equivalently the address range $[(h_0 - m_{\beta,W}) \cdot bps \cdot D, (h_0 + b_W + m_{\beta,W}) \cdot bps \cdot D - 1]$, where bps denotes bits per sample and D denotes channels per pixel. However, if just this memory range is dis-

³Whether these two transactions result in actual reads from global memory or cache hits depends on the specific microarchitecture of the GPU and the memory access pattern.

Figure 4.17: All threads in a block read lines cooperatively; each thread warp reads one or several 32-byte chunks. Useful data (in shades of green) may not be 32-byte aligned, so a small additional amount of data (bright red, not shown to scale) is read together with the desired data.

Figure 4.18: Four levels of the Z-order curve, an example of space-filling curve that serializes neighbouring positions in a 2-dimensional grid.

tributed among warps for them to read and $((h_0 - m_{\beta,W}) \cdot bps \cdot D) \bmod 32 \neq 0$, all of them will be 32-byte-misaligned and will result in two separate memory transactions. Therefore, it is best to align warps with the memory range to ensure that each memory address will be read only once; the range to be distributed among warps is $\left[\left\lfloor \frac{(h_0 - m_{\beta,W}) \cdot bps \cdot D}{32} \right\rfloor, \left\lceil \frac{(h_0 + b_W + m_{\beta,W}) \cdot bps \cdot D}{32} \right\rceil - 1 \right]$, as shown in figure 4.17.

Thus, we will read several whole 32-byte segments per line and each segment will be completely read by a single warp. By uniformly distributing among warps all the 32-byte segments in all the lines to be retrieved, we get the maximum transfer speed from global to shared memory. Then, each thread can proceed to use the shared data to compute the contributions of each reference pixel to the background model of its own pixel. This process is followed iteratively for each reference image.

Multilayered texture Two of the functional units of GPUs that support their traditional role in a computer graphics pipeline are the texture mapping and filtering units. Texture memory is not actually a separate tier of memory but rather an access mechanism to the global memory. Instead of the uncached access or linear caching that takes place (depending on devices) with regular accesses, the texture unit uses space-filling curves such as the Z-order curve [47], pictured in figure 4.18 as an example⁴, to cache the 2-dimensional neighborhood of a memory position instead of addresses lying on the same row.

Although traditionally textures were only 2-dimensional and the number of resident textures on the device was limited, hardware and software advances in the past few years have brought about the possibility of allocating 3D and multilayer 2D textures, which enabled us to implement the circular buffer structure that holds the reference image buffer using texture memory. Since our algorithm accesses reference data in the spatial vicinity of each pixel, this memory access is

⁴Manufacturers do not disclose details about the specific caching strategy, only functional promises about the texture cache being “optimized for 2D spatial locality” [121].

Figure 4.19: Mean per-pixel processing times using different spatial bandwidths ($\sigma_{\beta,H}, \sigma_{\beta,W}$) and depending on the selected thread block size ($b_H \times b_W$), with fixed appearance bandwidth and no selective update.

a perfect match and, although it does not guarantee that input pixels will only be read once, it proved indeed faster than the shared memory approach. Another beneficial side effect is that processing times are almost independent of the size of processing blocks (see dashed lines in figure 4.19), which improves the usability of the algorithm.

4.6.2.2 Lowering the cost of kernel evaluation

The proposed background model operates over a considerable amount of data and, as defined in equation (4.2), involves $D + 2$ (appearance characteristics + spatial dimensions) evaluations of the Gaussian function per reference datum (over 1000 evaluations per input pixel and image with usual parameters [142, 42]).

Unfortunately, exponentials are fairly expensive to evaluate, so this can be a bottleneck for the application. If the variance of the Gaussian is fixed and the range considered is discrete, such as is the case with 8-bit components, all possible evaluations of the Gaussian can be precomputed and stored in constant memory for fast access. However, if the variance is dynamically estimated the number of possible values is too big to efficiently store in a look-up table of exact values; the problem only gets worse if appearance components have a higher resolution, so an alternative method is required.

To solve this issue, we exploit that any Gaussian distribution can be expressed in terms of the standard normal distribution $\varphi = \mathcal{N}(0, 1)$:

$$f(x|\mu, \sigma) = \frac{1}{\sigma} \varphi\left(\frac{x - \mu}{\sigma}\right).$$

Furthermore, since the Gaussian has a horizontal asymptote in 0, it can be approximated by 0 for $x > c_G$, where c_G is a cutoff parameter that can be selected depending on the specific task at hand. This enables us to take n_G evenly spaced samples from $\mathcal{N}(0, 1)$ and build a look-up table of user-defined size, independent of variance or data resolution. Evaluation of any Gaussian thus becomes as simple as interpolating values of the look-up table. Figure 4.20 shows the absolute value of the relative error between the actual and interpolated value using nearest neighbor or linear interpolation for $n_G = 128$ and $c_G = 8$.

Employing linear interpolation significantly lowers the error for a given number of samples at a modest computational cost. Actually, since we have stored the look-up table as a unidimen-

Figure 4.20: Absolute value of the relative error between the actual and the interpolated values of the standard Gaussian function φ using nearest neighbour (blue) or linear interpolation (red) for $n_G = 128$ and $c_G = 8$.

sional texture instead of constant memory, the computational cost of the interpolation becomes negligible because linear interpolation is built-in in the circuitry of the texture filtering unit. A rigorous and in-depth mathematical analysis of the error of such approximations, as well as a detailed computational analysis, is given in chapter 5.

4.6.3 Particle filter

Many of the stages of the particle filter have been trivially implemented in a GPU (determining particle coverage, particle weights or prior foreground probability estimation). However, displacement of reference data requires significant changes in the implementation of the foreground model. When reference data are not moved from their original positions, access to data follows a regular pattern in which every thread reads reference data synchronously from the pixels that lie at the same relative position with respect to the current pixel, subject only to the condition, implemented with binary detection masks, that the potential reference pixel had been classified as foreground. On the other hand, when reference data are displaced, their original locations may be at arbitrary distances from the pixel that will use them as reference, and therefore a different mechanism must be implemented to access reference data. We evaluated three options: vector displacement maps, linear reference and multilayered reference images.

Vector displacement maps According to our model, we expect the object described by a set of measures $\{\mathbf{v}_i^n\}$ and its associated particle group summarized by its metaparticle $\mathbf{u}^n$ to move so that in the next frame it will roughly occupy the locations $\{(\mathbf{v}_i^n(1) + \dot{h}^n, \mathbf{v}_i^n(2) + \dot{w}^n)\}$. Therefore, we would set locations $(\lfloor \mathbf{v}_i^n(1) + \dot{h}^n \rfloor, \lfloor \mathbf{v}_i^n(2) + \dot{w}^n \rfloor)$ in the displacement map Δ^{n+1} to the vector values $(-\dot{h}^n, -\dot{w}^n)$. Then, when processing frame $n + 1$, each thread would look for reference data at the relative locations pointed at by the displacement map. This approach is obviously flawed because it would make impossible for reference data from two objects to be displaced onto the same location. Additionally, in order to retrieve reference data from frames older than n , displacement maps would need to be chained, and the chain would easily break near the edges of the (generally non-rigid) objects, decreasing the effective number of reference data.

Linear reference Since we already have the locations of all the pixels classified as foreground $\{\mathbf{v}_i^n\}$, it is trivial to join them with their labels and appearance information to produce

$(3 + D)$ -dimensional tuples per pixel $\mathbf{r}_i^n = (h_i^n, w_i^n, l_i^n, a_{i,1}^n, \dots, a_{i,c}^n)$, where l_i^n is the identifier of the particle group associated to the measure $\mathbf{v}_i^n$, and $a_{i,j}^n$ is the value of the j -th of D appearance components at pixel coordinates (h_i^n, w_i^n) of I^n . We can then join all reference pixels from the N_ϕ most recent frames to make up the whole reference data for the foreground $\mathbf{r} = \{\mathbf{r}^{n-f}\}_{f=0}^{N_\phi-1}$ and then apply the dynamic model so that the position of every reference pixel whose value l matches any of the active particle groups is updated to the predicted location for the next frame. Conceptually, this model works well, solves both of the problems that arise with vector displacement maps and has a reasonably compact representation in memory. However, its performance is poor because every execution thread in the foreground model must iterate over the whole set of reference pixels even though most of them are too far away to be relevant (or frequently all, since most of the image is typically background). Moreover, since memory transactions in CUDA devices are at least 32 bytes in size [163], even though execution threads will only need to read the location of the reference pixels to discard most of them, region identifiers and appearance coordinates will be also read.

Multilayered images An alternative arrangement consists in postprocessing and transforming the reference store $\mathbf{r}$ just defined, which we still use in order to manage the addition and removal of new and old reference data as well as the dynamic model, in order to dynamically create for each new frame a 3D image. In this image, the appearance components of elements of $\mathbf{r}$ with the same (h, w) coordinates will be stacked along the z dimension of the image in pixels with coordinates $(h, w, \cdot)$. The size of the 3D image along the z dimension is not known beforehand and neither is the number of valid pixels for each (h, w) combination; consequently, we use an auxiliary 2D image of unsigned integers to record the number of valid samples at each location. Storing reference data as a 3D image results in very sparse data and incurs a significant penalty because the whole 3D image must be regenerated and transmitted to the GPU every frame. However, this penalty is offset by the fact that the number of iterations that execution threads in the foreground model must perform is dramatically reduced (most threads can readily see that there are no reference data in their vicinity and exit early, and those that do have reference data can selectively read only what they need), and effective memory bandwidth increased because the memory access pattern is much more regular: neighbouring threads need neighbouring data, increasing cache hit rate and execution coherence.

For each frame, the threads of the foreground model iterate along the z axis of the 3D reference image, only reading those pixels that satisfy equation (4.12) and contain valid data on the current layer according to the auxiliary image; when no valid pixels are found at a given depth, all the reference data has been read and the loop ends.

4.6.3.1 Selective analysis and regions of interest

Processing the image block by block comes naturally in CUDA because its computational model is based on 1D, 2D or 3D blocks of threads, whose geometry is typically chosen to fit the data. In the case of NPM, 2D blocks of threads are a good match to rectangular blocks of the input images and even without selective analysis, images are processed block by block. The only changes that selective analysis imposes over the previous NPM implementation is that blocks should have a meaningful size in relation to the objects that need to be detected: if they are too big, too many pixels will have to be analyzed; if they are too small, too many analysis iterations will need to be carried out for the algorithm to converge.

Figure 4.21: Average F -score across the LASIESTA database using only the background model for different numbers of bins for dynamic width switching.

4.7 Results

The proposed background subtraction strategy has been tested on a very large variety of indoor and outdoor sequences containing critical challenges for moving object detection. These sequences have been extracted from three databases:

- SABS dataset⁵: It is a very popular database that was proposed in [28] and has been recently used by many authors to test their algorithms [33, 143, 70]. It is composed by synthetic sequences to evaluate seven different challenging situations. All the images have a ground truth mask associated. To emulate the noise introduced by real camera sensors, the Gaussian noise was added to the pixel values of sequences. Moreover, they have been created using a raytracing technology with global illumination to simulate scenes with realistic lighting.
- STAR dataset⁶: It is composed by nine real video sequences recorded in different indoor and outdoor environments [95]. In contrast to SABS database, the ground truth for each of these sequences consists of a subset of 20 images randomly selected. However, the challenges in this database have made it one of the most used by authors to test their moving object detection approaches [70, 45, 99].
- LASIESTA database⁷: It is a new database that stands out among other databases due to the great amount of challenges it contains and because it is the only existing database with real videos that are fully anotated at both pixel-level and object-level.

The quality of the detections for the SABS and LASIESTA databases has been measured using the harmonic mean of the recall ($r = \frac{cd}{cd+md}$) and precision ($p = \frac{cd}{cd+fd}$) or F -score ($F = 2 \frac{r \cdot p}{r+p}$), where cd is the number of correct detections, fd is the amount of false detections and md is the number of misdetections; for the STAR dataset we have used the average similarity score ($sim = \frac{cd}{cd+fd+md}$), which is the measure reported in the literature for this database.

All the experiments have been carried out on an NVIDIA GTX 580 GPU with 1.5 GiB of RAM, coupled with an Intel Core i7-2600 with 16 GiB RAM, under Ubuntu 14.04 64-bit and CUDA 4.2.1.

In section 4.7.1, the adequate selection of the parameters described along the document is discussed. Then, in section 4.7.2 we analyze the performance of the proposed strategy in different cases. Finally, in section 4.7.3 we compare the results provided by our strategy with those obtained with other state-of-the-art alternatives.

⁵<http://www.vis.uni-stuttgart.de/index.php?id=sabs>

⁶http://perception.i2r.a-star.edu.sg/bk_model/bk_index.html

⁷<http://www.gti.ssr.upm.es/data/LASIESTA>

Dataset	BG			BG + FG			BG + FG + RoI		
	Recall	Precision	F	Recall	Precision	F	Recall	Precision	F
SABS	0.7289	0.6921	0.6761	0.8585	0.6470	0.7243	0.8563	0.6550	0.7284
STAR	0.4078	0.8231	0.5387	0.8258	0.7480	0.7738	0.8196	0.7525	0.7735
LASIESTA	0.7922	0.7530	0.7548	0.9195	0.6980	0.7815	0.9181	0.7346	0.8051
Average	0.6430	0.7561	0.6565	0.8679	0.6977	0.7599	0.8647	0.7140	0.7690

 Table 4.1: Recall, precision and F values obtained in three key stages of the proposed strategy.

4.7.1 Parameter selection

To reduce the influence of shadows and reflected light in the detections, both foreground and background models are obtained using the appearance vector described in [44], which is composed by the chromaticity ($Rn = \frac{R}{R+G+B}$, $Gn = \frac{G}{R+G+B}$) and the module of the spatial brightness gradient $|\nabla s|$, where $s = R + G + B$.

In the case of background modeling it is only necessary to determine the number of the reference images and the spatial width assigned to the Gaussian kernels. The first one must be large enough to model the cyclical background changes along the sequences. We have set it to $T_\beta = 200$ for the sequences in both STAR and LASIESTA databases; and as $T_\beta = 650$ for the sequences in SABS dataset, since these sequences include much larger background cycles. Regarding the spatial width of the kernels, the performed experiments have shown that most false detections are avoided using a neighborhood of 8-connected pixels ($\sigma_{\beta,H}^2 = \sigma_{\beta,W}^2 = (\sqrt{2}/3)^2$). Since larger spatial widths significantly increase the computational cost [42], such neighborhood has been used for all the analyzed databases. The number of bins used for the appearance width estimation could be specified as discussed in the bandwidth switching paragraph in section 4.3.1.1 but, as figure 4.21 shows, a fine estimation is not necessary: even very coarse partitions yield reasonable results and there is no reason to use more than twenty bins. Consequently, we have used this value across all experiments.

In the case of the foreground model, it is also necessary to set the number of reference images and the spatial width of the kernels. We have set the former as $T_\phi = 10$, which is enough in any sequence, since the foreground does not typically exhibit cyclical changes as the background may do. The spatial width of the kernels has been set slightly higher than that of the background to allow for some imprecision in the motion estimation, $\sigma_{\phi,H}^2 = \sigma_{\phi,W}^2 = (2/3)^2$, and does not usually need tuning. Finally, for the sequences in LASIESTA and STAR we have set a fixed appearance width of 0.02 for both chromaticities and the spatial brightness gradient, which is sufficiently larger than the typical values for the background noise, which are in the order of 10^{-3} in the set of appearance components we have used. However, for the sequences in SABS this width has been slightly increased to 0.10, since it contains many illumination changes that induce persistent false detections if set too low.

4.7.2 Performance of the proposed strategy

To validate each of the stages of the proposed strategy, we have analyzed the results and their associated computational costs in three cases:

- BG: Modeling only the background.
- BG + FG: Modeling both background and foreground.
- BG + FG + RoI: Modeling both background and foreground and using the RoI module.

Figure 4.22: (a) Original images. Results with: (b) only background model (BG); (c) adding foreground model (BG + FG); (d) adding the RoI analysis (BG + FG + RoI).

Dataset	BG	BG + FG	BG + FG + RoI
SABS	10	9	25
STAR	30	18	28
LASIESTA	30	21	42

Table 4.2: Average frames per second for three key configurations of the proposed strategy; all three datasets have been processed with CIF (352×288) resolution.

Figure 4.23: Some representative results obtained with the proposed detection strategy on SABS database. (a) Original images. (b) Obtained detections.

Table 4.1 summarizes the quality measures obtained in the three cases and some representative results are illustrated in figure 4.22. It can be observed that by adding the foreground modeling, the amount of misdetections is drastically reduced: the detections are more compact (less red pixels in figure 4.22.b) and the average recall increases significantly. Adding the RoI module (figure 4.22.c) eliminates small false detections (black pixels) whereas the amount of correct detections is maintained, resulting in a slight increase of the precision and similar recall.

Table 4.2 shows average speed (frames per second) achieved in the three aforementioned configurations in the three evaluated databases. Note that, in order to be fair, all measurements are referred to the fractions of the sequences where all the models have been fully initialized and the system has reached a stationary regime, starting from image $I^{T_\beta+1}$, since previous images are much faster to evaluate simply because they have fewer data to process. Evaluation times of the background model depend on the number of used reference images, T_β , as expected (it is approximately constant because the amount of data and the number of operations is the same irrespectively of the contents of the images). The cost of the foreground model, on the other hand, depends on the fraction of foreground present in the scene. Finally, by adding the RoI module the cost of the background model is drastically reduced, allowing real-time operation in all the evaluated databases.

4.7.3 Comparison with other detection strategies

4.7.3.1 SABS dataset

Some representative results obtained with the proposed strategy on the sequences of SABS are illustrated in figure 4.23, and the achieved F -score values and those reported by several alternative approaches are shown in table 4.3. The test procedure for this database allows tuning one parameter per sequence [28]. However, to prove the high usability of our proposal, we have used the same set of parameters for all of them (although this puts us at a disadvantage compared to other algorithms). Additionally, it must be mentioned that the use of post-processing steps is not allowed in the evaluation of this database. Therefore, the regularization step applied by some algorithms (e.g., in [70]) has been disabled to perform this comparison.

Although the proposed nonparametric strategy does not provide the best results in every category, it shows consistent performance in all of them; consequently, it obtains the best average F -score. Additionally, it is remarkable that in very dark scenarios (e.g. Darkening, Light switch and Noisy night) it is significantly better than previous approaches, since it is able to detect objects that are barely distinguishable from the background to the naked eye.

In some sequences (e.g. Dynamic Background) the achieved F -score may seem not high

Method	Basic	Dynamic background	Bootstrap	Darkening
1995-McFarlane [107]	0.614 (11)	0.482 (11)	0.541 (09)	0.496 (09)
1999-Stauffer [148]	0.800 (02)	0.704 (06)	0.642 (06)	0.404 (11)
2000-Oliver [123]	0.635 (10)	0.552 (10)	-	0.300 (13)
2000-McKenna [108]	0.522 (13)	0.415 (12)	0.301 (12)	0.484 (10)
2003-Li 1 [94]	0.766 (06)	0.641 (08)	0.678 (04)	0.704 (03)
2004-Kim [87]	0.582 (12)	0.341 (13)	0.318 (11)	0.342 (12)
2006-Zivkovik [170]	0.768 (04)	0.704 (05)	0.632 (07)	0.620 (07)
2008-Maddalena 1 [99]	0.766 (05)	0.715 (03)	0.495 (10)	0.663 (05)
2009-Barnich [9]	0.761 (07)	0.711 (04)	0.658 (05)	0.678 (04)
2013-Shimada [143]	0.723 (09)	0.623 (09)	0.708 (02)	0.577 (08)
2013-Cuevas [44]	0.759 (08)	0.729 (02)	0.557 (08)	0.653 (06)
2014-Haines [70]	0.836 (01)	0.827 (01)	0.717 (01)	0.736 (02)
Proposed	0.779 (03)	0.694 (07)	0.680 (03)	0.778 (01)

Method	Light switch	Noisy night	Camouflage	No camouflage	Average
1995-McFarlane [107]	0.211 (11)	0.203 (09)	0.738 (10)	0.785 (09)	0.509 (11)
1999-Stauffer [148]	0.217 (09)	0.194 (10)	0.802 (05)	0.826 (03)	0.574 (09)
2000-Oliver [123]	0.198 (12)	0.213 (08)	0.802 (04)	0.824 (04)	0.503 (12)
2000-McKenna [108]	0.306 (06)	0.098 (11)	0.624 (12)	0.656 (12)	0.426 (13)
2003-Li 1 [94]	0.316 (05)	0.047 (12)	0.768 (08)	0.803 (06)	0.590 (07)
2004-Kim [87]	-	-	0.776 (07)	0.801 (07)	0.527 (10)
2006-Zivkovik [170]	0.300 (07)	0.321 (06)	0.820 (02)	0.829 (02)	0.624 (04)
2008-Maddalena 1 [99]	0.213 (10)	0.596 (02)	0.793 (03)	0.811 (05)	0.590 (06)
2009-Barnich [9]	0.268 (08)	0.271 (07)	0.741 (09)	0.799 (08)	0.614 (05)
2013-Shimada [143]	0.335 (04)	0.475 (03)	-	-	0.574 (08)
2013-Cuevas [44]	0.572 (02)	0.336 (05)	0.736 (11)	0.744 (11)	0.636 (03)
2014-Haines [70]	0.499 (03)	0.346 (04)	0.848 (01)	0.851 (01)	0.708 (02)
Proposed	0.637 (01)	0.705 (01)	0.780 (06)	0.774 (10)	0.728 (01)

Table 4.3: F -score values of several background subtraction strategies evaluated on the SABS database. Algorithm ranks are given by the numbers in parentheses. The last column contains the average F -score for all the tests.

Figure 4.24: (a) Image from SABS database where a moving object is partially occluded by a tree. (b) Detection obtained with the proposed strategy (similar color notation to that used in previous figures). (c) Binary mask resulting from the detection.

Method	Bootstrap	Curtain	Fountain	Shopping Mall
1999-Stauffer [148]	0.3838 (5)	0.7580 (4)	0.6854 (3)	0.5363 (5)
2004-Li 2 [95]	0.3079 (6)	0.1841 (7)	0.0999 (7)	0.5209 (6)
2007-Culibrk [45]	0.4779 (4)	0.7368 (5)	0.4636 (5)	0.5696 (4)
2008-Maddalena 1 [99]	0.6019 (3)	0.8178 (3)	0.6554 (4)	0.6677 (1)
2013-Cuevas [44]	0.1709 (7)	0.2317 (6)	0.4150 (6)	0.2589 (7)
2014-Haines [70]	0.6024 (2)	0.8203 (2)	0.7049 (1)	0.6522 (2)
Proposed	0.6410 (1)	0.8412 (1)	0.7006 (2)	0.6415 (3)
Method	Switch light	Trees	Water surface	Average
1999-Stauffer [148]	0.6519 (3)	0.0757 (7)	0.7948 (4)	0.5551 (5)
2004-Li 2 [95]	0.1554 (7)	0.1596 (6)	0.0667 (7)	0.2135 (7)
2007-Culibrk [45]	0.6276 (4)	0.5256 (4)	0.7540 (5)	0.5936 (4)
2008-Maddalena 1 [99]	0.6489 (2)	0.6960 (3)	0.8247 (3)	0.7018 (3)
2013-Cuevas [44]	0.2886 (6)	0.3779 (5)	0.3921 (6)	0.3050 (6)
2014-Haines [70]	0.5794 (5)	0.7567 (1)	0.9090 (1)	0.7178 (2)
Proposed	0.6628 (1)	0.7037 (2)	0.9011 (2)	0.7274 (1)

Table 4.4: Similarity values of several detection strategies evaluated on the STAR database. Algorithm ranks are given by the numbers in parentheses. The last column contains the average similarity for all the tests.

Figure 4.25: Some representative results obtained with the proposed detection strategy on the STAR database. (a) Original images. (b) Obtained detections.

enough. However, as it can be seen in figure 4.24, our spatiotemporal modeling strategy partially “fills” voids due to the occlusion of the tree leaves and results in a subjectively correct detection, even if it is objectively wrong according to the per-pixel provided ground-truth.

4.7.3.2 STAR dataset

The similarity values obtained for this dataset are shown in table 4.4 and some representative results are illustrated in figure 4.25. We have discarded two of the nine test sequences of this database that were encoded with very low quality, since the proposed background modeling has not been designed to model pixel variations due to coding artifacts. Unlike SABS, this database allows using post-processing stages. Additionally, it allows tuning the algorithms to improve the quality of the results on each sequence. Although all the proposals shown in table 4.4 except [70] take advantage of this, again, our results have been obtained with a single set of parameters.

The results in figure 4.25 show that our strategy is able to provide successful detections in all the challenges proposed in STAR: complex dynamic backgrounds, illuminations, changes,

Figure 4.26: Some representative results of the proposed detection strategy on the LASIESTA database. (a) Original images. (b) Obtained detections.

moving objects remaining static, etc. Additionally, as can be seen in table 4.4, it achieves the best average similarity.

4.7.3.3 LASIESTA dataset

This database is composed of many more sequences than any of the two previous databases, and all of them are fully annotated. For our experiments, we have only selected those sequences without camera motion (a total of twenty sequences). Unlike in previous databases, the evaluation of the sequences in LASIESTA should be carried out without training period. All algorithms, including ours, have been run with a single set of parameters throughout the sequences.

The F -score values obtained for this dataset are shown in Table 4.5, and some representative results are illustrated in figure 4.26. Unlike with the other two databases, where two or three of the evaluated methods achieve the best results in each sequence, in LASIESTA there are up to five winning strategies (see the results highlighted in the table), due to the wide variety of challenges contained. Again, even if our strategy does not take the top spot in many sequences, it shows reasonable and consistent performance, taking the best overall F -score. Our weakest results (I_OC_01 or O_SU_02) come in sequences containing reflections or hard shadows cast by the moving objects since, unlike other researchers (e.g., [170, 100]) we do not apply any processing stage focused on shadow removal.

4.8 Conclusions

We have presented a high-quality nonparametric moving object detection strategy based on kernel density estimation techniques, implemented in real-time in a GPU. The proposed strategy features: robust spatio-temporal models of both the background and the foreground; two alternative algorithms to automatically estimate the appropriate per-pixel kernel bandwidths for the background model; a selective update scheme that allows the user to easily control the adaptation rate of the background model to enhance the detection of slowly moving objects; a Bayesian classifier that is able to combine models with different spatial bandwidths using the marginal distributions obtained conditioning both the foreground and background models at each particular location, thereby preventing persistent false detections; a novel low-level tracking system based on a particle filter, capable of dealing with a variable and unknown number of moving regions, in order to update the positions of the foreground reference data over time and provide prior probability estimations for the classifier; and a selective analysis algorithm that automatically

Method	I	SI	O1	I	SI	O2	I	CA	O1	I	CA	O2	I	OC	O1	I	OC	O2	I	IL	O1
1999-Stauffer [148]	0.8247	(6)		0.8013	(6)		0.8609	(5)		0.7542	(5)		0.9118	(4)		0.8089	(7)		0.3608	(5)	
2006-Zivkovik [170]	0.9118	(4)		0.8990	(2)		0.9094	(3)		0.7546	(4)		0.9880	(1)		0.9135	(3)		0.1648	(7)	
2008-Maddalena 1 [99]	0.8928	(5)		0.8465	(3)		0.9532	(1)		0.7394	(6)		0.9803	(2)		0.8466	(6)		0.8533	(2)	
2012-Maddalena 2 [100]	0.9559	(2)		0.9409	(1)		0.8416	(7)		0.8731	(1)		0.9573	(3)		0.9508	(2)		0.1898	(6)	
2013-Cuevas [44]	0.8143	(7)		0.7576	(7)		0.8424	(6)		0.6296	(7)		0.8274	(6)		0.8781	(5)		0.7966	(3)	
2014-Haines [70]	0.9622	(1)		0.8130	(5)		0.9220	(2)		0.8656	(2)		0.8920	(5)		0.9526	(1)		0.8861	(1)	
Proposed	0.9208	(3)		0.8403	(4)		0.9062	(4)		0.7826	(3)		0.7013	(7)		0.8600	(4)		0.6452	(4)	
Method	I	IL	O2	I	MB	O1	I	MB	O2	I	BS	O1	I	BS	O2	O	CL	O1	O	CL	O2
1999-Stauffer [148]	0.2505	(6)		0.8213	(6)		0.6881	(5)		0.3754	(7)		0.3816	(7)		0.8847	(6)		0.8189	(7)	
2006-Zivkovik [170]	0.3135	(5)		0.9321	(4)		0.8015	(3)		0.5472	(3)		0.5195	(4)		0.9303	(3)		0.8226	(6)	
2008-Maddalena 1 [99]	0.3750	(4)		0.8473	(5)		0.6761	(7)		0.4023	(5)		0.4465	(5)		0.8985	(5)		0.8547	(5)	
2012-Maddalena 2 [100]	0.2312	(7)		0.9728	(2)		0.8517	(2)		0.4015	(6)		0.4021	(6)		0.9657	(1)		0.9760	(1)	
2013-Cuevas [44]	0.7864	(2)		0.7779	(7)		0.6797	(6)		0.5065	(4)		0.6607	(2)		0.9280	(4)		0.8995	(4)	
2014-Haines [70]	0.8122	(1)		0.9816	(1)		0.7064	(4)		0.6285	(2)		0.7333	(1)		0.6946	(7)		0.9588	(2)	
Proposed	0.6523	(3)		0.9543	(3)		0.9204	(1)		0.7132	(1)		0.6156	(3)		0.9508	(2)		0.9045	(3)	
Method	O	RA	O1	O	RA	O2	O	SN	O1	O	SN	O2	O	SU	O1	O	SU	O2	Average		
1999-Stauffer [148]	0.7411	(7)		0.8003	(7)		0.7298	(4)		0.4785	(3)		0.6169	(6)		0.8083	(5)		0.6859	(7)	
2006-Zivkovik [170]	0.8586	(1)		0.8980	(3)		0.5206	(6)		0.2402	(5)		0.5426	(7)		0.8775	(3)		0.7173	(6)	
2008-Maddalena 1 [99]	0.8252	(4)		0.8588	(6)		0.6977	(5)		0.4595	(4)		0.7467	(3)		0.8562	(4)		0.7528	(4)	
2012-Maddalena 2 [100]	0.8353	(3)		0.9591	(1)		0.9093	(2)		0.7116	(1)		0.8742	(1)		0.8843	(2)		0.7842	(2)	
2013-Cuevas [44]	0.7462	(6)		0.8699	(5)		0.8214	(3)		0.0895	(6)		0.6527	(5)		0.8074	(6)		0.7386	(5)	
2014-Haines [70]	0.8225	(5)		0.9590	(2)		0.3054	(7)		0.0426	(7)		0.8115	(2)		0.9021	(1)		0.7826	(3)	
Proposed	0.8453	(2)		0.8886	(4)		0.9317	(1)		0.6256	(2)		0.6774	(4)		0.7669	(7)		0.8051	(1)	

Table 4.5: F-score values of several detection strategies evaluated on the LASIESTA database. Algorithm ranks are given by the numbers in brackets. The last column contains the average for all the tests

selects regions of interest in the input images, yielding equivalent results at a fraction of the computational cost.

Due to their considerable memory and processing requirements, KDE-based techniques have been generally eschewed in favour of simpler, cheaper alternatives. However, we have demonstrated by providing a real-time implementation of the full proposal that makes coordinated use of both CPU and GPU, that parallel processors such as those found in regular GPUs are a very good match with KDE techniques and make them perfectly current and valid.

The proposed strategy has been tested on a large variety of sequences from three databases containing many challenges for motion detection, and compared against many state-of-the-art methods. The obtained results demonstrate real-time performance of our approach and its superior ability to adapt and consistently deliver state-of-the-art results in a wide variety of scenarios, the best results on average, without requiring extensive tuning of its parameters.

Chapter 5

Optimal Piecewise Linear Function Approximation for GPU-based Applications

The approximation of complex nonlinear systems by simpler piecewise linear representations is a recurrent and attractive task in many applications since the resulting simplified models have lower complexity, fit into well established tools for linear systems and are capable of representing arbitrary nonlinear mappings. Examples include, among others, complexity reduction for finding the inverse of nonlinear functions [74, 153], distortion mitigation techniques such as predistorters for power amplifier linearization [32, 7], the approximation of nonlinear vector fields obtained from state equations [10], the obtention of approximate solutions in simulations with complex nonlinear systems [150], or the search for canonical piecewise linear representations in one and multiple dimensions [84].

In the last decades, the main efforts in piecewise linearization have been devoted both to find approximations of multidimensional functions from a mathematical standpoint and to define circuit architectures implementing them (see, for example, [27] and references therein). In the one-dimensional setting, a simple and common linearization strategy consists in building a linear interpolant between samples of the nonlinear function over a uniform partition of its domain. Such a polygonal (i.e., continuous piecewise linear) interpolant may be further optimized by choosing a better partition of the domain according to the minimization of some error measure. This is a sensible strategy in problems where there is a constraint on the budget of samples allowed in the partition.

Hence, in spite of the multiple benefits derived from modeling with piecewise linear representations, a proper selection of the interval partitions and/or predefining the number of partitions is paramount for a satisfactory performance. Some researchers [153] use cross-validation based approaches to select such a number of pieces within a partition. In other applications, the budget of pieces may be constrained by an internal design requirement (speed, memory or target error) of the approximation algorithm or by some external condition.

Even though advances in computer performance have generally made direct evaluation of ever more complex functions generably affordable, there are applications, for instance nonparametric background subtraction (see chapter 4), where it constitutes a significant fraction of the computation. Therefore, reducing the cost of function evaluations may have a direct impact in the overall performance of this kind of applications. GPU vendors are aware of this recurrent

computing problem and provide hardware implementations of common transcendental functions in Special Function Units (SFUs) [96]; indeed, we can find in the literature examples of successful use of these hardware facilities [151]. However, they only implement a handful of functions and users cannot choose their numerical precision, which is reduced [121] with respect to the reference implementations of standard mathematical libraries.

We propose a novel, fast, and practical method to evaluate any mathematical function continuously differentiable within a known interval. Our contributions include, based on error equalization, a nearly optimal design of two types of piecewise linear approximations (linear interpolant and orthogonal projection) in the L_2 norm under the constraint of a large budget of evaluation subintervals N . Moreover, we provide asymptotically tight bounds for the approximation errors of both piecewise linear representations, improving upon existing ones. Specifically, in addition to the $O(N^{-2})$ convergence rate typical of these approximations, we quantify their interpolation constants: $1/\sqrt{120}$ for the linear interpolant and a further $\sqrt{6}$ improvement factor in case of the orthogonal projection. The obtained error relations are parameterized by the number of segments used to represent the complex (nonlinear) function, hence our approach allows the user to estimate the errors given N or, conversely, estimate the required N to achieve a target approximation error.

We also propose an efficient implementation of the technique in a modern GPU by exploiting the fixed-function interpolation routines present in its texture units to accelerate the computation while leaving the rest of the GPU (and, of course, the CPU) free to perform other tasks; this technique is even faster than using SFUs [96].

Although the initial motivation of the developed method was to improve the performance of nonparametric foreground segmentation strategies, it must be noted that it can be also used in many other scientific fields such as computer vision [63, 66] or signal processing [165, 139, 62], where the evaluation of continuous mathematical functions constitutes a significant computational burden.

5.1 Piecewise Linear Modelization

5.1.1 Related Work

In many applications, rigorous evaluation of complex mathematical functions is not practical because it takes too much computational power. Consequently, this evaluation is carried out approximating them by simpler functions such as (piecewise-)polynomial ones. Piecewise linearization has been used as an attractive simplified representation of various complex nonlinear systems [150]. The resulting models fit into well-established tools for linear systems and reduce the complexity of finding the inverse of nonlinear functions [74, 153]. They can also be used to obtain approximate solutions in complex nonlinear systems, for example, in Mixed-Integer Linear Programming (MILP) models [138]. Some efforts have been devoted as well to the search for canonical representations in one and multiple dimensions [84, 85] with different goals such as black box system identification, approximation or model reduction.

Previous attempts to address the problem considered here (optimal function piecewise linearization) include [11, 59, 74, 64], the latter two in the context of nonlinear dynamical systems. In [11] an iterative multi-stage procedure based on dynamical programming is given to provide a solution to the problem on sequences of progressively finer 2-D grids. In [74] the piecewise linear approximation is obtained by using the evolutionary computation approach such as genetic algorithm and evolution strategies; the resulting model is obtained by minimization of a sampled version of the mean squared error and it may not be continuous. In [64] the problem is addressed using a hybrid approach based on curve fitting, clustering and genetic algorithms.

Figure 5.1: Top: fifth-degree polynomial $f(x) = (x + 4)(x + 2)(x + 1)(x - 1)(x - 3)$ and two continuous piecewise linear (CPWL) approximations, the orthogonal projection $P_T f$ and the linear interpolant $\pi_T f$; bottom: corresponding absolute approximation errors (magnified by a $5\times$ factor).

Figure 5.2: Hat functions constitute a basis of V_T . Since all the basis functions are zero outside $I = [a, b]$, the basis functions associated with boundary nodes are only *half hats*.

Here we address the problem from a less heuristic point of view, using differential calculus to derive a more principled approach. Our method solely relies on standard numerical integration techniques, which take few seconds to compute, as opposed to recursive partitioning techniques such as [59], which take significantly longer.

5.1.2 Basic Results in Piecewise-Linear Interpolation and Least-Squares Approximation

In this section we summarize the basic theory behind piecewise linear functions [54] and two such approximations to real-valued functions: interpolation (section 5.1.2.1) and projection (section 5.1.2.2), both pictured in figure 5.1 along with their absolute approximation error with respect to an example fifth-degree polynomial function.

In essence, a piecewise function over an interval $I = [a, b]$ is a partition of I into a set of N subintervals $T = \{I_i\}_{i=1}^N$, where $I_i = [x_{i-1}, x_i]$ | $a = x_0 < x_1 < \dots < x_N = b$, and a set of N functions $f_i(x)$, one for each subinterval I_i . In particular we are interested in continuous piecewise linear (CPWL) functions, which means that all the $f_i(x)$ are linear and $f_i(x_i) = f_{i+1}(x_i) \forall i = \{1, \dots, N-1\}$. CPWL functions of a given partition T are elements of a vector space V_T : the addition of such functions and/or multiplication by a scalar yields another CPWL function defined over the same subintervals. A useful basis for the vector space V_T is formed by the set of *hat functions* or *nodal basis functions* $\{\varphi_i\}_{i=0}^N$, pictured in figure 5.2 and defined in

general by the formula

$$\varphi_i(x) = \begin{cases} (x - x_{i-1})/(x_i - x_{i-1}), & x \in [x_{i-1}, x_i] \\ (x - x_{i+1})/(x_i - x_{i+1}), & x \in [x_i, x_{i+1}] \\ 0, & x \notin [x_{i-1}, x_{i+1}]. \end{cases} \quad (5.1)$$

The basis functions φ_0 and φ_N associated to the boundary nodes x_0 and x_N are only *half hats*.

These basis functions are convenient since they can represent any function v in V_T by just requiring the values of v at its nodal points, $v(x_i)$, in the form

$$v(x) = \sum_{i=0}^N v(x_i) \varphi_i(x). \quad (5.2)$$

5.1.2.1 Linear Interpolation

The piecewise linear interpolant $\pi_T f \in V_T$ of a continuous function f over the interval I can be defined in terms of the basis just introduced:

$$\pi_T f(x) = \sum_{i=0}^N f(x_i) \varphi_i(x). \quad (5.3)$$

While this CPWL approximation is trivial to construct, and may be suitable for some uses, it is by no means the best possible one. Crucially, $\pi_T f(x) \geq f(x) \forall x \in I$ for any function f that is convex in I . Depending on the application in which this approximation is used, this property could skew the results. However, as we will see in section §5.2, the linear interpolant is useful to analyze other possible approximations. It is also at the heart of the trapezoidal rule for numerical integration.

5.1.2.2 Orthogonal Projection onto Vector Space V_T

Let the usual inner product between two square-integrable (L_2) functions in the interval I be given by

$$\langle u, v \rangle = \int_I u(x) v(x) dx. \quad (5.4)$$

Then, the vector space V_T can be endowed with the above inner product, yielding an inner product space. As usual, let $\|u\| = \sqrt{\langle u, u \rangle}$ be the norm induced by the inner product, and let $d(u, v) = \|u - v\|$ be the distance between two functions u, v . The orthogonal projection of the function f onto V_T is the function $P_T f \in V_T$ such that

$$\langle f - P_T f, v \rangle = 0 \quad \forall v \in V_T. \quad (5.5)$$

Since $P_T f \in V_T$, it can be expressed in the nodal basis $\{\varphi_i\}_{i=0}^N$ by

$$P_T f(x) = \sum_{i=0}^N c_i \varphi_i(x), \quad (5.6)$$

where the coefficients c_i solve the linear system of equations $Mc = b$. The $(N+1) \times (N+1)$ matrix $M = (m_{ij})$ has entries $m_{ij} = \langle \varphi_i, \varphi_j \rangle$, and vector b has entries given by $b_i = \langle f, \varphi_i \rangle$. The Gramian matrix M is tridiagonal and strictly diagonally dominant. Therefore, the system

has exactly one solution c , which can be obtained efficiently using the Thomas algorithm (a particular case of Gaussian elimination) [156].

$P_T f$ is the element in V_T that is closest to f in the sense given by the aforementioned L_2 distance d , as we recall next. For any $w \in V_T$,

$$\begin{aligned} \|f - P_T f\|^2 &= \langle f - P_T f, f - w + w - P_T f \rangle \\ &= \langle f - P_T f, f - w \rangle + \langle f - P_T f, w - P_T f \rangle \\ &\stackrel{(5.5)}{=} \langle f - P_T f, f - w \rangle, \end{aligned}$$

where we used, as per equation (5.5), that the vector $(w - P_T f) \in V_T$. Next, applying the Cauchy-Schwarz inequality,

$$\|f - P_T f\|^2 = \langle f - P_T f, f - w \rangle \leq \|f - P_T f\| \|f - w\|,$$

and so $\|f - P_T f\| \leq \|f - w\|$, i.e.,

$$d(f, P_T f) \leq d(f, w) \quad \forall w \in V_T, \tag{5.7}$$

with equality if $w = P_T f$. This makes $P_T f$ most suitable as an approximation of f under the L_2 norm.

5.2 Finding the Optimal Partition

As just established, for a given vector space of CPWL functions V_T , $P_T f$ is the function in V_T whose L_2 distance to f is minimal. However, the approximation error $\|f - P_T f\|$ does not take the same value for every possible V_T ; therefore, we would like to find the optimal partition T^* (or a reasonable approximation thereof) corresponding to the space V_{T^*} in which the approximation error is minimum,

$$\|f - P_{T^*} f\| \leq \|f - P_T f\|.$$

This is a difficult optimization problem: in order to properly specify $P_T f$ in equation (5.6) and measure its distance to f , we need to solve the linear system of equations $Mc = b$, whose coefficients depend on the partition itself, which makes the problem symbolically intractable. Numerical algorithms could be used but it is still a challenging problem.

Let us examine the analogous problem for the interpolant function $\pi_T f(x)$ defined in section 5.1.2.1. Again, we would like to find the optimal partition T^* corresponding to the space V_{T^*} in which the approximation error is minimum,

$$\|f - \pi_{T^*} f\| \leq \|f - \pi_T f\|.$$

Albeit not as difficult as the previous problem, because $\pi_T f$ is more straightforward to define, this is still a challenging non-linear optimization problem. Fortunately, as it is shown next, a good approximation can be easily found under some asymptotic analysis.

In the rest of this section we investigate in detail the error incurred when approximating a function f by the interpolant $\pi_T f$ and the orthogonal projection $P_T f$ defined in section §5.1 (see figure 5.1). Then we derive an approximation to the optimal partition (section 5.2.2) that serves equally well for $\pi_T f$ and $P_T f$ because, as we will show (section 5.2.1), their approximation errors are roughly proportional under the assumption of a sufficiently large number of intervals.

Figure 5.3: Function f in the subinterval $I_i = [x_{i-1}, x_i]$ and two linear approximations. On the left, the linear interpolant $\pi_T f$ given by equation (5.8); on the right, a general linear segment line_i given by equation (5.10). Δy_j is a signed distance with respect to $f(x_j)$.

5.2.1 Error in a Single Interval

5.2.1.1 Linear Interpolant

First, let us consider the error incurred when approximating a function f , twice continuously differentiable, by its linear interpolant in an interval (figure 5.3).

Theorem 5.1. *The L_2 error between a given function f and its linear interpolant*

$$\pi_T f(x) = f(x_{i-1})(1 - \delta_i(x)) + f(x_i)\delta_i(x), \quad (5.8)$$

with $\delta_i(x) = (x - x_{i-1})/h_i$, in the interval $I_i = [x_{i-1}, x_i]$, of length $h_i = x_i - x_{i-1}$, is bounded:

$$\|f - \pi_T f\|_{L_2(I_i)} \leq \frac{1}{\sqrt{120}} |f''|_{\max} h_i^{5/2}, \quad (5.9)$$

where $|f''|_{\max} = \max_{\eta \in I_i} |f''(\eta)|$.

Proof. See section §A.1. □

This result admits the following intuitive explanation: the error measures the deviation of f from being straight (linear), and this is directly related to the convexity/concavity of the function, thus the presence of the term $|f''|_{\max}$ to bound the amount of bending.

The other interesting part of equation (5.9) is the $h_i^{5/2}$ dependence with respect to the interval size or the local density of knots. Other works in the literature use the weaker bound that does not require carrying out integration [46]: $\|f - \pi_T f\|_{L_2(I_i)} \leq C |f''|_{\max} h_i^2$, for some constant $C > 0$.

5.2.1.2 Best Linear Approximation

Let us now characterize the error of the orthogonal projection $P_T f$. We do so in two steps: first we compute the minimum error of a line segment in the interval I_i , and then we use an asymptotic analysis to approximate the error of the orthogonal projection.

Stemming from equation (5.8), we can write any (linear) segment in I_i as

$$\begin{aligned} \text{line}_i(x) &= (f(x_{i-1}) + \Delta y_{i-1})(1 - \delta_i(x)) \\ &\quad + (f(x_i) + \Delta y_i)\delta_i(x), \end{aligned} \quad (5.10)$$

where Δy_{i-1} and Δy_i are extra degrees of freedom (pictured in figure 5.3) with respect the interpolant $\pi_T f$ that allow the line segment to better approximate the function f in I_i . By computing the optimal values of Δy_{i-1} and Δy_i we obtain the following result.

Theorem 5.2. *The minimum squared L_2 error between a given function f and a line segment in an interval $I_i = [x_{i-1}, x_i]$, of length $h_i = x_i - x_{i-1}$, adopts the expression*

$$\begin{aligned} \min \|f - \text{line}_i\|_{L_2(I_i)}^2 &= \frac{h_i^5}{120} (f''(\eta))^2 \\ &\quad - \frac{h_i^5}{144} \left((f''(\eta_0))^2 + (f''(\eta_1))^2 - f''(\eta_0)f''(\eta_1) \right). \end{aligned} \quad (5.11)$$

for some η, η_0, η_1 in (x_{i-1}, x_i) .

Proof. See section §A.2. □

Corollary 5.3. *If I_i is sufficiently small so that f'' is approximately constant within it, say f''_{I_i} , then*

$$\min \|f - \text{line}_i\|_{L_2(I_i)}^2 \approx \frac{h_i^5}{720} (f''_{I_i})^2. \quad (5.12)$$

Proof. Substitute $f''(x) \approx f''_{I_i}$ for $x = \{\eta, \eta_0, \eta_1\}$ in equation (5.11). □

The segments in $P_T f$ may not strictly satisfy equation (5.11) because $P_T f$ must also be continuous across segments. However, as the size of the intervals h_i become smaller (due to a finer partition T of the interval I , i.e., the number of segments N in T increases) we may approximate $P_T f$ in I_i by the best (independently-optimized) segment line_i and, consequently, use Corollary 5.3 to yield the following result.

Theorem 5.4. *The squared L_2 error between a given function f and its orthogonal projection $P_T f$ in a small interval $I_i = [x_{i-1}, x_i]$, of length $h_i = x_i - x_{i-1}$, is*

$$\|f - P_T f\|_{L_2(I_i)}^2 \approx \frac{h_i^5}{720} (f''_{I_i})^2. \quad (5.13)$$

where f''_{I_i} is the approximately constant value that f'' takes within the interval I_i .

Proof. See section §A.3. □

In the same asymptotic situation, the linear interpolant defined in equation (5.9) gives a bigger squared error (see equation (A.4)) by a factor of six,

$$\|f - \pi_T f\|_{L_2(I_i)}^2 \approx \frac{h_i^5}{120} (f''_{I_i})^2. \quad (5.14)$$

5.2.2 Approximation to the Optimal Partition

Now we give a procedure to compute a suboptimal partition of the target interval I and then derive error estimates for both $\pi_T f$ and P_T on such a partition and the uniform one.

5.2.2.1 Procedure to Obtain a Suboptimal Partition

Let us consider a partition T of $I = [a, b]$ with subintervals $I_i = [x_{i-1}, x_i]$, $i = 1, \dots, N$. A suboptimal partition for a given N is one in which every subinterval has approximately equal contribution to the total approximation error [46, 40], which implies that regions of f with higher convexity are approximated using more segments than regions with lower convexity. Let

us assume N is large enough so that f'' is approximately constant in each subinterval and therefore the bound given by equation (5.9) is tight. Consequently,

$$|f''|_{\max} h_i^{5/2} \approx C, \quad (5.15)$$

for some constant $C > 0$, and the lengths of the subintervals (local knot spacing [40]) should be chosen $h_i \propto |f''|_{\max}^{-2/5}$, i.e., smaller intervals as $|f''|_{\max}$ increases. Hence, the local knot distribution or density is

$$lkd(x) \propto |f''(x)|^{2/5}, \quad (5.16)$$

so that more knots of the partition are placed in the regions with larger magnitude of the second derivative.

Then, the proposed approximation $T^{(*)}$ to the optimal partition is as follows: $x_0 = a$, $x_N = b$, and take knots $\{x_i\}_{i=1}^{N-1}$ given by

$$F(x_i) = i/N, \quad (5.17)$$

where the monotonically increasing function $F : [a, b] \rightarrow [0, 1]$ is

$$F(x) = \int_a^x (f''(t))^{2/5} dt / \int_a^b (f''(t))^{2/5} dt. \quad (5.18)$$

In this procedure, pictured in figure 5.4, the range of $F(x)$ is divided into N contiguous sub-ranges of equal length, and the values of x_i are given by the abscissas corresponding to the endpoints of the sub-ranges.

5.2.2.2 Error Estimates

With this partition, we can further estimate approximation error bounds in the entire interval based on those of the subintervals.

Theorem 5.5. *The approximation error of the linear interpolant $\pi_{T^{(*)}} f$ in the partition $T^{(*)}$ given by equation (5.17) in $I = [a, b]$ is*

$$\|f - \pi_{T^{(*)}} f\| \lesssim \frac{1}{N^2 \sqrt{120}} \left(\int_a^b (f''(t))^{2/5} dt \right)^{5/2}. \quad (5.19)$$

Proof. The total squared error for any partition T is the sum of squared errors over all subintervals I_i , and by equation (5.9),

$$\|f - \pi_T f\|^2 \leq \sum_{i=1}^N \frac{1}{120} \left(|f''|_{\max} h_i^{5/2} \right)^2, \quad (5.20)$$

which, under the condition of error equalization of equation (5.15) for the proposed partition $T^{(*)}$, becomes

$$\|f - \pi_{T^{(*)}} f\|^2 \leq \sum_{i=1}^N \frac{1}{120} C^2 = \frac{1}{120} C^2 N. \quad (5.21)$$

To compute C , let us sum $|f''|_{\max}^{2/5} h_i \approx C^{2/5}$ over all intervals I_i and approximate the result using the Riemann integral:

$$C^{2/5} N \stackrel{(5.15)}{\approx} \sum_{i=1}^N |f''|_{\max}^{2/5} h_i \approx \int_a^b (f''(t))^{2/5} dt, \quad (5.22)$$

Figure 5.4: Graphical summary of the proposed knot placement technique. Top: local knot density (equation (5.16)) obtained from input function f (example in figure 5.1); middle: cumulative knot distribution function F given equation (5.18) and knots given by the abscissas corresponding to a uniform partition of the range of F , as expressed by equation (5.17); bottom: approximation of f by CPWL interpolant $\pi_{T^{(*)}}f$ with $N = 31$ (32 knots). In this suboptimal partition, knots are distributed according to the amount of local convexity/concavity of f given by the lkd in the middle plot. Hence, fewer knots are placed around the zeros of the lkd , which correspond to the less steep regions of F .

whose right hand side is independent of N . Substituting equation (5.22) in equation (5.21) gives the desired approximate bound (equation (5.19)) for the error in the interval $I = [a, b]$. $\square$

Since the approximation error of $P_T f$ in each interval is roughly proportional to that of $\pi_T f$, as shown in equation (5.13) and equation (5.14), the partition given by equation (5.17) is also a very good approximation to the optimal partition for $P_T f$ as the number of subintervals N increases. This is briefly stated next.

Theorem 5.6. *The approximation error of the orthogonal projection $P_{T^{(*)}}f$ in the partition $T^{(*)}$ given by equation (5.17) in $I = [a, b]$ is*

$$\|f - P_{T^{(*)}}f\| \approx \frac{1}{\sqrt{6}} \|f - \pi_{T^{(*)}}f\|. \quad (5.23)$$

Both CPWL approximations ($\pi_T f$ and $P_T f$) converge to the true function f at a rate of at least $O(N^{-2})$ (equation (5.19) and equation (5.23)).

We use a similar procedure to derive an estimate error bound for the uniform partition T_U that can be compared to that of the optimized one.

Theorem 5.7. *The approximation error of the linear interpolant $\pi_{T_U}f$ in the uniform partition T_U of the interval $I = [a, b]$ is*

$$\|f - \pi_{T_U}f\| \approx \frac{(b-a)^2}{N^2 \sqrt{120}} \|f''\|. \quad (5.24)$$

Proof. For T_U , we may substitute $h_i = (b-a)/N$ in equation (5.20) and approximate the result

Algorithm 5.1 (CPWL function evaluation). Given as input the description of a CPWL $v \in V_T$ by means of the the nodal values $\mathbf{v_samples} = v(x_0), \dots, v(x_N)$ at the knots $\mathbf{x_samples} = x_0, \dots, x_N$ of the partition T , and an abscissa $x^* \in [x_0, x_N]$, return $v(x^*)$.

- 1 Determine the subinterval I_i that contains x^* : $i \mid x_{i-1} \leq x^* < x_i$
 - 2 Find the fractional distance δ_i to the left endpoint of I_i : $\delta_i \leftarrow (x^* - x_{i-1}) / (x_i - x_{i-1})$
 - 3 Return the interpolated value $(1 - \delta_i) v(x_{i-1}) + \delta_i v(x_i)$
-

using the Riemann integral,

$$\|f - \pi_{T_U} f\| \stackrel{(5.20)}{\leq} \frac{h_i^2}{\sqrt{120}} \left(\sum_{i=1}^N |f''|_{\max}^2 h_i \right)^{1/2} \approx \frac{(b-a)^2}{N^2 \sqrt{120}} \|f''\|.$$

□

Thus, we can estimate how much we can expect to benefit from optimizing a partition by simply dividing equation (5.19) by equation (5.24). Since equation (5.24) also shows a $O(N^{-2})$ convergence, the profit will not depend on N (assuming N is large enough).

5.3 Computational Analysis

The implementation of the approximations previously discussed is straightforward (Algorithm 5.1) and we only need to distinguish two different cases depending on whether the partition T of I has all subintervals of equal width or not. Although their values are different, $\pi_T f(x)$ and $P_T f(x)$ are qualitatively the same from the computational viewpoint.

Let us discuss the general case that T is non-uniform. Line 1 in Algorithm 5.1 implies searching in the ordered array $\mathbf{x_samples}$ for the right index i . Since the other steps in the algorithm do not depend on the size of the input vectors, Line 1 is the dominant step and makes its run-time complexity $O(\log N)$. In the particular case that T is uniform, no search is needed to determine the index: $i \leftarrow \lfloor \frac{x^* - x_0}{x_N - x_0} N \rfloor + 1$ and the fractional part $\delta_i \leftarrow \frac{x^* - x_0}{x_N - x_0} N + 1 - i$. Thus, the run-time complexity of the uniform case is $O(1)$. There is also no need to store the full array $\mathbf{x_samples}$ but only its endpoints x_0 and x_N , roughly halving the memory needed for a non-uniform partition of the same number of intervals.

Consequently, approximations based on a uniform partition are expected to perform better in usual CPUs or GPUs, computationally-wise, than those based on non-uniform partitions. However, optimizing the partition might lead, depending on the specific objective function, to a reduction in the memory requirements for a desired approximation error. If memory constraints are relevant or hardware-aided search routines were available in the future, optimized partitions could become practical.

5.4 Implementation in a GPU

The proposed algorithm is simple to implement either in a CPU or in a GPU. However, there are some implementation details that help further optimize the performance in a GPU.

Modern GPUs implement a Single Instruction, Multiple Data (SIMD) execution model in which multiple threads execute exactly the same instructions (each) over different data. If two threads scheduled together in the same execution group or *warp* follow different branches in

a conditional statement, both branches are not executed in parallel but sequentially, thereby halving throughput. In the case of T being non-uniform, the (binary) search performed to find the index i could lead to divergent paths for different threads evaluating $v(x)$ for different values of x , thus reducing throughput.

Since the purpose of the proposed algorithm is to save computation at run-time, it is reasonable to assume that at least N , and possibly `x_samples` and `v_samples` too, have been determined at compile time. Then, we can particularize the search code in compile time, unrolling the loop needed to implement the binary search and, most importantly, replacing conditional statements with a fixed number of invocations of the ternary operator, which is implemented in the GPU with a single instruction and no code divergence [126].

As explained in section 2.4.3, accesses to memory in a GPU need, in general, to be arranged so that neighbouring threads access neighbouring addresses in memory. This allows the host interface of the GPU to coalesce operations from several threads into fewer transactions bigger in size. In the described algorithm, if each thread is evaluating $v(x)$ for different values of x , it is not guaranteed that the memory access pattern complies with this restriction. Although some devices provide caching and isolate the programmer from this issue, not all do.

However, there is a case in a regular computer graphics pipeline in which a similar problem arises. Texture data need to be accessed many times in a short period and it is generally impossible to ensure that neighbouring pixels in screen need to read neighbouring texels. Consequently, most GPUs do cache texture data and even have dedicated cache layers for it. We therefore store the arrays of samples in texture memory rather than in global memory to benefit from texture caches.

The use of the texture units to access our samples provides another important benefit. In the usual computer graphics pipeline, GPUs need to constantly filter texture data. In order to efficiently perform that task, texture filtering units (TFUs) are equipped with hardware fixed-function interpolation routines that include linear interpolation [96]. Therefore, we can use TFUs to implement line 3 in Algorithm 5.1, thus leaving the regular cores free to simultaneously execute other instructions. Results in the next section confirm that this approach is indeed faster than performing interpolation “manually”.

5.5 Experimental Results

In this section we apply the proposed linearization algorithm to several functions of interest in computer vision and other scientific areas. The section starts with the analysis of the Gaussian function and demonstrates the proposed technique on an image processing application. The section then analyzes two other sample nonlinear functions of interest: the Lorentzian function and the Bessel function of the first kind.

5.5.1 Gaussian Function

The Gaussian function $f(x) = \exp(-x^2/2)/\sqrt{2\pi}$ is widely used in many applications in every field of science, but it is of particular interest to us in the context of foreground segmentation in video sequences using spatio-temporal nonparametric background models (see chapter 4). To estimate these models, the Gaussian function needs to be evaluated approximately 1300 times per pixel and input image, or around 2 billion times per second at modest video quality (CIF, 352×288 pixels at 15 fps). Therefore it is crucial to lower the computing time of the Gaussian function. In the performed experiments it was enough to approximate the function in the interval $x \in [0, 8]$ to achieve the required quality.

Figure 5.5: L_2 distance for different approximations to the Gaussian function $f(x) = \exp(-x^2/2)/\sqrt{2\pi}$ over the interval $x \in [0, 8]$.

Table 5.1: Net mean per-evaluation execution times (in picoseconds) for the Gaussian function.

	Number of points ($N + 1$)	32	64	128	256	512
CPU	exact function using <code>expf</code> from <code><math></code>	13710				
	uniform partition	1750 - 1780				
	optimized partition	8900	11230	13830	16910	20120
GPU	exact function using <code>expf</code> from <code><math></code>	33.3				
	fast function using <code>__expf</code> from the SFU	14.2				
	uniform partition, manual interpolation	19.7 - 19.8				
	uniform partition, hardware interpolation	7.8 - 7.9				
	optimized partition, manual interpolation	142.0	161.9	181.8	203.1	224.1
	optimized partition, hardware interpolation	122.7	142.9	163.2	188.3	211.1

L_2 distances between the Gaussian function and the approximations described in previous sections are shown in figure 5.5. It reports actual distances as well as the approximate and tight upper bounds to the distances, given by theorems 5.5 to 5.7. It can be observed that the L_2 distances between $\pi_T f$ and f using the uniform and optimized partitions agree well with Results 5.5 to 5.7. All curves in figure 5.5 have similar slope to that of equation (5.19), i.e., their convergence rate is $O(N^{-2})$, and the ratio between the distances corresponding to $P_T f$ and $\pi_T f$ is approximately the value $1/\sqrt{6}$ that stems from theorem 5.4 and equation (5.14).

Mean per-evaluation processing are reported in table 5.1, both in the CPU (sequentially, one core only) and in the GPU. All execution times have been measured in a computer equipped with an Intel Core i7-2600K processor clocked at 3.8 GHz, 16 GiB RAM and an NVIDIA GTX 580 GPU. We have designed the experiments to ensure that all different strategies achieve a high occupancy of the GPU. In addition, we have made use of on-chip shared memory and registers to reduce global memory accesses to accurately recreate a scenario where the execution is purely compute-bounded. Measured times are based on the execution of the same generic code, first evaluating the identity function to get the baseline execution time, then substituting the function

Figure 5.6: Mean per-evaluation execution times of all the proposed variants on a GPU (in picoseconds). The graph clearly shows the $O(1)$ and $O(\log N)$ dependencies on the number of intervals N for uniform and non-uniform partitions, respectively.

to be measured for the identity function to measure the gross execution time. Execution times reported in table 5.1 are the net result of subtracting the baseline execution time from the gross execution time of each of the measured functions. Note that the proposed technique only provides an advantage in situations where the algorithm is compute-bound, since we are essentially substituting computation for memory accesses, although if frequently used most of these access should result in cache hits. Conversely, in bandwidth-bound scenarios, it may make more sense to stick to the SFU.

We compare the proposed algorithm against the exact Gaussian implemented using the exponential function of the C standard library (CPU or GPU) and against a fast Gaussian implemented using the hardware-accelerated low-precision exponential function from the SFU of the GPU [121]. In either hardware (CPU or GPU), there are no separate measurements for $\pi_T f(x)$ and $P_T f(x)$ because they only differ in value but not in their implementation. As was expected from the computational analysis, execution times for the uniform partition are constant, whereas they are heavily dependent on N for the optimized partition (see figure 5.6). In the latter case, it is worth mentioning that the GPU fares specially badly because the search for the correct interval induces frequent pipeline stalls. As explained in section §2.3, CPUs are designed to better exploit instruction-level parallelism and they implement techniques such as data forwarding, which allow the scheduler to dispatch instructions that use operands still being computed inside the pipeline, thereby reducing stalls.

In the CPU case, the optimized partition approach is faster than the C standard library implementation if $N < 128$, but the speed-up gain is smaller than that of the uniform partition approach, which is 8 times faster than the C standard library (regardless of N).

In the GPU case, we have also measured the proposed algorithm both with and without the texture filtering unit (last four rows of table 5.1) to prove that the former option is indeed faster than coding the interpolation explicitly, thanks to its dedicated circuitry. The proposed strategy, using a uniform partition (7.9 ps), solidly outperforms both the exact (33.3 ps) and fast SFU (14.2 ps) Gaussians, by approximate speed-up factors of $\times 4$ and $\times 2$, respectively, even though

Table 5.2: Mean per-pixel processing times (in nanoseconds) for the nonparametric background modeling technique.

Uniform partition, hardware interpolation	286
Fast Gaussian using <code>__expf</code> from the SFU	324
Exact Gaussian using <code>expf</code> from <code><cmath></code>	391

the latter is hardware-accelerated.

These results have been obtained in purely compute-bounded conditions and therefore applications whose bottleneck is clearly not in the access to memory can expect to benefit the most from the proposed strategy; however, as is the case in any optimisation of a computer program, only testing in the real application can determine for sure whether one method is faster than other. In a real application such as our initial target application, there may exist limitations in memory bandwidth to access operands. Moreover, many resources such as shared memory, cache and hardware registers are being used for other tasks too, which can affect the execution time of each of the different implementation strategies.

As an example of a real application, we have used our proposed technique in the nonparametric background modeling method described in section 4.3.1, which is a prime candidate for optimization because a huge amount of Gaussian kernels need to be evaluated. Table 5.2 shows mean per-pixel (in order to make measurements resolution-independent) processing times for the three best performing options in section §5.5. Processing times are not proportional to those in table 5.1 because the model needs to do many other things aside from evaluating Gaussians, and access to reference data is bandwidth-limited. However, our proposed technique still outperforms any of the alternatives: the exact Gaussian by 27% and the fast SFU Gaussian by 12%, while the final segmentation results of the method are the same in all three cases.

5.5.2 Lorentzian Function

The performance of the approximation technique has also been tested on other functions. For example, the Lorentzian function with peak at x_0 and width γ is

$$\mathcal{L}(x; x_0, \gamma) = \frac{1}{\pi} \frac{\gamma}{(x - x_0)^2 + \gamma^2}. \quad (5.25)$$

As a probability distribution, equation (5.25) is known as the Cauchy distribution. It is an important function in Physics since it solves the differential equation describing forced resonance. In such case, x_0 is the resonance frequency and γ depends on the damping of the oscillator (and is inversely proportional to the Q factor, a measure of the sharpness of the resonance).

L_2 distances between $\mathcal{L}(x; 0, 1)$ and the CPWL approximations described in previous sections are reported in figure 5.7 for the interval $x \in [0, 20]$. The measured errors agree well with the predicted approximate error bounds, showing the expected $O(N^{-2})$ trend.

We have measured the mean per-evaluation execution times of the exact function both in the CPU (576 ps) and in the GPU, both using regular and reduced-precision accelerated division (19.5 ps and 9.2 ps, respectively). The evaluation times of our proposed strategy coincide with those of the Gaussian function (table 5.1) because the processing time of the CPWL approximation does not depend on the function values.

As expected from the lower complexity of this function (compared to that of section §5.5.1), which only involves elementary arithmetic operations, the advantage of our proposal vanishes in the CPU because the operations needed to manually perform interpolation, together with necessary fetching of the two values from memory, are actually more than what the direct evaluation

Figure 5.7: L_2 distance for different CPWL approximations to the standard Cauchy distribution $f(x) = 1/((1+x^2)\pi)$ in the interval $x \in [0, 6]$.

requires. However, note that in the GPU our proposal still remains competitive because these operations are carried out by the dedicated circuitry of the texture unit.

5.5.3 Bessel Function

Approximating the Bessel function of the first kind $J_0(x)$ is more challenging than approximating the Gaussian or Lorentzian (bell-shaped) functions because it combines both nearly flat and oscillatory parts (see figure 5.8). L_2 approximation errors for $J_0(x)$ using CPWL functions are presented in figure 5.9 for the interval $x \in [0, 20]$. For $N \geq 63$ the measured errors agree well with the predicted approximate error bounds, whereas for $N < 63$ the measured errors slightly differ from the predicted ones (specifically in the optimized partition) because in these cases the scarce number of linear segments does not properly represent the oscillations.

A sample optimized partition and the corresponding CPWL interpolant $\pi_{T(*)}f$ is also represented in figure 5.8. The knots of the partition are distributed according to the local knot density (see figure 5.8, top), accumulating in the regions of high oscillations (excluding the places around the zeros of the lkd).

This function is more complex to evaluate in general because it does not have a closed form. However, our approximation algorithm works equally well on it. We have measured the mean per-evaluation execution times in the CPU (39 ns using the POSIX extensions of the GNU C Library and 130 ns—only double-precision provided—using the GNU Scientific Library) and in the GPU (78 ps using the CUDA [121] standard library). We have also measured the execution time of the first term in the asymptotic expansion [26, equation (9.57a)], valid for $x \gg 1/4$, $J_0(x) = \sqrt{\frac{2}{\pi x}} [\cos(x - \frac{\pi}{4}) + O(\frac{1}{x})]$. In the GPU the evaluation takes 42 ps using reduced-precision accelerated functions from the SFU for the cosine and multiplicative inverse of the square root. Despite this approximation having a non-customizable error (decreasing as x increases) and using the fastest available implementation (SFU) of the operations involved, our strategy still outperforms it by a sizable margin. This clearly illustrates that the more complex the function to be evaluated is, the greater the advantage of our proposal.

Figure 5.8: Suboptimal partition for the Bessel function $f(x) = J_0(x)$ in the interval $x \in [0, 20]$. Top: local knot density (lkd) corresponding to f ; bottom: CPWL interpolant $\pi_{T*} f$ with $N = 63$ (64 knots) overlaid on function f .

Figure 5.9: L_2 distance for different CPWL approximations to the Bessel function of the first kind $f(x) = J_0(x)$ in the interval $x \in [0, 20]$.

5.6 Conclusions

We have developed a fast method to numerically evaluate any continuous mathematical function in a given interval using simpler continuous piecewise linear (CPWL) functions and the built-in routines present in the texture filtering units of modern GPUs. Our technique allows real-time implementation of demanding applications that make heavy use of such mathematical functions in computer vision or any other field.

For this purpose, we analyzed the CPWL approximations given by the linear interpolant and the L_2 orthogonal projection of a function. We carried out a detailed error analysis in the L_2 distance to seek a nearly optimal design of both approximations. In the practical asymptotic case of a large number of subintervals N , we used error equalization to achieve a suboptimal design (partition $T^{(*)}$) and derived a tight bound on the approximation error for the linear interpolant, showing a $O(N^{-2})$ convergence rate that was confirmed by experimental results. The L_2 orthogonal projection can only but improve upon the results of the linear interpolant, resulting in a gain factor of $\sqrt{6}$.

We discussed the computational complexity and the implementation in a GPU of the numerical evaluation of both CPWL approximations. Our experimental results show that our technique can outperform both the quality and the computational cost of previous similar approaches. In particular, the fastest strategy consists of using the texture units in the GPU to evaluate either of the CPWL approximations defined over a uniform partition. This is normally faster than performing exact function evaluations, even when using the reduced-precision accelerated implementation of the SFUs in a GPU, or evaluating the proposed linear approximations without the assistance of the texture units. In practice, the number of subintervals N to be considered for representing the nonlinear function can be decided on its own or based on other considerations such as a target approximation error, speed or memory constraints.

Although mathematically sound, the strategies based on a suboptimal (non-uniform) partition are not practical to implement in current CPU/GPU architectures due to the high cost incurred to find the subinterval that contains the given point of evaluation. Nevertheless, this opens future research paths to explore practical implementations of such approaches using specialized hardware.

Chapter 6

Conclusions and future research

In the last few years, a decisive change in the design of processors and computers has taken place. Single-core processor performance has stopped to increase at the pace it had the previous decades because of physical constraints and also increasingly due to market pressure, which demands energy-efficient solutions that can be used in mobile devices. Now, the design trend goes toward increasing the degree of parallelism, placing more and more cores in each processor, thereby requiring current and future algorithms to be (re)designed to reduce strictly sequential stages to a minimum in order to take full advantage of the available hardware.

Following the trend of energy-efficient parallel design, a new device class has emerged in the last decade: graphics processing units, formerly application-specific devices, are now usable for general-purpose computing as massively parallel devices, featuring hundreds of processing cores optimised for raw numeric performance. Though these devices are still at a relatively early stage of development and present a number of restrictions in their use that do not make them necessarily the fastest choice for every task, we think they are too powerful a tool to ignore and that every algorithm designer ought to be aware of their potential, strengths and weaknesses.

In this thesis, we have explored two compute-intensive application domains in computer vision—free viewpoint video and background subtraction—making use of the GPU to achieve real-time performance, and proposed a general technique to significantly speed up the evaluation of complex functions over a specified domain on a GPU.

We have presented a feasibility study on view interpolation using depth-image-based rendering techniques from just two cameras with depth information when they are significantly apart from each other in a non-parallel, approximately convergent configuration. Usual applications of DIBR techniques in 3D TV are limited to short-baseline parallel camera configurations, which allow the use of simpler techniques such as forward mapping DIBR and reduce the dimensionality of the correspondences. The camera configuration we have studied calls for a costlier technique (backward mapping DIBR) and a depth-filtering stage in order to eliminate cracks in the virtual depth maps. However, thanks to the parallelisation of the problem on a GPU, we have achieved interactive control and real-time performance.

While carrying out the study, we have identified a potential problem in the coding of depth images, which arises when depth values are referred to a global reference system. This is mostly harmless when numeric precision is not limited, but in practice depth information is usually coded as a grayscale image and therefore severely quantized. The concurrence of both depth quantization and global depth reference proves inadequate for unrestricted camera configurations, which becomes apparent when locally referred depth information yields excellent results. This problem is not apparent in the usual parallel camera configuration because the depth information

is the same whether it is globally referred to one of the cameras or to each camera.

We have also presented a background subtraction algorithm for video sequences, based on nonparametric KDE techniques. These techniques offer great flexibility and adaptability to a wide range of scenarios, but have seen relatively little use in the literature due to their sizable memory and computational cost.

The most relevant theoretical issues we have found applying KDE modelling to background subtraction are: selection of a correct kernel bandwidth; too fast learning rate when slow-moving objects are present; and persistent false detections derived from combining foreground and background models because of their significant mismatch in spatial distribution of reference data. In order to solve each of these issues, we have proposed: two alternative low-cost automatic kernel bandwidth estimators; a selective update method that allows the user to control the learning rate of the background model in a straightforward way; and an alternative Bayesian classifier that decouples appearance and spatial information of both foreground and background models by conditioning them on each spatial location.

The most significant practical problem with KDE modelling derives from the huge number of kernels that must be evaluated. Some authors have mitigated this problem by using fixed bandwidths and/or simpler kernel functions, at the cost of reducing the quality of their detections. Instead, we have tackled the issue with three proposals: reducing the per-datum cost of kernel evaluation using linear approximations implemented using the texture units of the GPU; reducing the spatial bandwidth used to model the foreground (and therefore the number of samples to consider) thanks to a multi-region, low level tracking system based on particle filters, that aims at compacting the foreground reference data around the current position of each moving object; and finally reducing even more the number of samples that need to be analysed through a selective analysis strategy that leverages the aforementioned tracking system to automatically determine regions of interest.

We have implemented the nonparametric model and all the proposed improvements on a GPU, resulting in a high-quality background subtraction system that runs in real time and has proven its versatility on a wide variety of scenes from several databases against many other state-of-the-art alternatives from the literature.

Recognising that the fast evaluation of functions with a high computational cost can be useful for many other applications in computer vision or any other scientific field, we have carried out an exhaustive analysis, from both the mathematical and the computational points of view, of our proposal for approximating arbitrary functions with continuous piecewise linear functions. We have shown that using the texture filtering units for this task may be an attractive option to squeeze out more performance from the GPU in applications with high arithmetic intensity, because it frees up the computing cores for executing other instructions meanwhile.

To sum up, we have found the use of the GPU an invaluable ally to achieve high throughput in the applications we have covered in this thesis and also in other applications not included herein. For now, development for these devices is relatively involved and it is necessary to know many details of the underlying architecture to write efficient code. In addition, we have suffered all kinds of failures, due to both software (compiler bugs) and hardware (random bit flipping). However, we are sure that compilers will become better at abstracting the programmer from device-specific quirks in years to come; in fact, during the elaboration of this thesis we have witnessed successive changes in hardware and software capabilities that enable better development methodology (e.g. true object-oriented programming).

However, the computing model is by nature quite rigid and applications whose data does not conform to regular geometries suffer heavy penalties. In particular, we have found cumbersome to transform data from one geometry to another, especially if data sizes are not known beforehand. While it is now possible to allocate memory in runtime from within the GPU, the need for absolute

coherence in the memory management structures of a global heap make its use not so efficient. While there are mechanisms that allow inter-thread communication, they are economical only between threads within the same block. Inter-block communication is very inefficient because it requires atomic instructions on global memory, which result in drastic throughput reduction and make the use of dynamic memory less attractive.

We believe that the implementation of some high-speed inter-block communication facility, maybe in the same spirit of intra-block shared memory, however small, would be a valuable addition to the architecture. However, since general purpose computing on GPUs is, for the moment, a byproduct of their main function as graphics processors, maybe this will not happen in the near future. However, even so, GPUs are already a formidable addition to the toolbox of any computer scientist and their use will only increase in years to come.

6.1 Future work

We have shown that background subtraction based on KDE techniques is versatile and can be made practical through a parallel implementation and a number of improvements we have proposed. However, there are many other possible outstanding enhancements:

- **Smart temporal subsampling:** One of the main characteristics of nonparametric methods like ours is that their model is made only from past observations we can still remember. On the one hand, this is desirable because it makes for a more robust model, since past errors do not feed back and cause future errors. On the other hand, this is not so desirable because, the farther back we want to remember from the past, the more memory and computational cost we incur. In general, the idea is that the reference buffer should only be as long as necessary to include instances of all the cyclic states the background can adopt. However, in some sequences where cycles are very long (e.g. SABS), this requires storing and processing many reference images, despite the fact that many of them are redundant. A keyframe detection algorithm would help subsampling the reference images and dispense with the need to exhaustively store so many images.
- **Detection of static moving objects:** In some application scenarios such as surveillance systems, detecting moving objects is not enough. In these cases, it is also desirable to be able to automatically detect objects that *stop* moving (e.g. abandoned luggage, foreign objects on roads or wrongly parked vehicles). The selective update scheme we have proposed for the background model offers users easy control over how long a moving object needs to remain still before the background model begins to absorb it; therefore it constitutes an excellent starting point for more complex strategies that combine several such models with different update delays as the inputs of a finite state machine [104] that yields richer classifications than a background subtraction system alone.
- **Illumination change detection:** Some scenarios demand opposite behaviour from our models: on the one hand, a nonparametric model with blind update is extremely fast adapting to any change, including sudden illumination changes, but on the other hand, it is equally fast absorbing foreground objects, hence resulting in non compact detections. We have proposed a selective update mechanism that solves the latter situation but, at the same time, delays the adaptation of the model against sudden illumination changes. Although our proposal is still ahead of many others in this respect, it would be desirable to add a module for specifically detecting illumination changes to eliminate or reduce the tradeoff we have described.

Regarding our proposal for fast approximation of functions, since the texture filtering units also support two-dimensional interpolation, it might be interesting to extend the idea to higher-order functions ($\mathbb{R}^2 \rightarrow \mathbb{R}$) if there are applications that could benefit from such approach. Extension to even higher dimensions is probably not practical due to the excessive data size that would be necessary to adequately sample the function.

Appendix A

Proofs from chapter 5

A.1 Proof of theorem 5.1 (Linear Interpolant Error)

Recall one of the theorems on interpolation errors [34]: let f be a function in $C^{n+1}[a, b]$, and let p be a polynomial of degree n or less that interpolates the function f at $n + 1$ distinct points $x_0, x_1, \dots, x_n \in [a, b]$. Then, for each $x \in [a, b]$ there exists a point $\xi_x \in [a, b]$ for which

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) \prod_{k=0}^n (x - x_k). \quad (\text{A.1})$$

In the interval I_i , the linear interpolant $\pi_T f$ is given by equation (5.8). Since $\pi_T f$ interpolates the function f at the endpoints of I_i , we can apply equation (A.1) (with $n = 1$); hence, the approximation error solely depends on f'' and x , but not on f or f' :

$$f(x) - \pi_T f(x) = -\frac{1}{2} f''(\xi_x) (x - x_{i-1})(x_i - x). \quad (\text{A.2})$$

Let us integrate the square of equation (A.2) over the interval I_i ,

$$\|f - \pi_T f\|_{L_2(I_i)}^2 = \int_{I_i} \frac{(f''(\xi_x))^2}{4} ((x - x_{i-1})(x_i - x))^2 dx.$$

Next, to simplify the previous integral, let us use the first mean value theorem for integration, which states that if $u : [A, B] \rightarrow \mathbb{R}$ is a continuous function and v is an integrable function that does not change sign on the interval (A, B) , then there exists a number $\eta \in (A, B)$ such that

$$\int_A^B u(x)v(x) dx = u(\eta) \int_A^B v(x) dx. \quad (\text{A.3})$$

Since $(x - x_{i-1}) \geq 0$ and $(x_i - x) \geq 0$ for all $x \in I_i$, let us apply equation (A.3) to compute

$$\begin{aligned} \|f - \pi_T f\|_{L_2(I_i)}^2 &= \frac{1}{4} (f''(\eta))^2 \int_{I_i} ((x - x_{i-1})(x_i - x))^2 dx \\ &= \frac{h_i^5}{120} (f''(\eta))^2, \end{aligned} \quad (\text{A.4})$$

for some $\eta \in (x_{i-1}, x_i)$. Finally, if $|f''|_{\max} = \max_{\eta \in I_i} |f''(\eta)|$, it is straightforward to derive the L_2 error bound of equation (5.9) from the square root of equation (A.4).

A.2 Proof of theorem 5.2 (Line Segment Minimum Error)

The approximation error corresponding to the line segment defined by equation (5.10) is

$$f(x) - \text{line}_i(x) = f(x) - \pi_T f(x) - (\pi_T \Delta y)(x) \quad (\text{A.5})$$

where, by analogy with the form described in equation (5.8) of $\pi_T f$, we defined $(\pi_T \Delta y)(x) = \Delta y_{i-1}(1 - \delta_i(x)) + \Delta y_i \delta_i(x)$.

The proof proceeds by computing the optimal values of Δy_{i-1} and Δy_i that minimize the squared L_2 error over the interval I_i :

$$\begin{aligned} \|f - \text{line}_i\|_{L_2(I_i)}^2 &\stackrel{(\text{A.5})}{=} \|f - \pi_T f\|_{L_2(I_i)}^2 + \|\pi_T \Delta y\|_{L_2(I_i)}^2 \\ &\quad - 2 \langle f - \pi_T f, \pi_T \Delta y \rangle_{L_2(I_i)}. \end{aligned} \quad (\text{A.6})$$

The first term is given in equation (A.4). The second term is

$$\|\pi_T \Delta y\|_{L_2(I_i)}^2 = \frac{h_i}{3} ((\Delta y_{i-1})^2 + (\Delta y_i)^2 + \Delta y_{i-1} \Delta y_i),$$

and the third term is, applying equation (A.3) and the change of variables $t = \delta_i(x)$ to evaluate the resulting integrals,

$$\begin{aligned} &-2 \langle f - \pi_T f, \pi_T \Delta y \rangle_{L_2(I_i)} \\ &= \Delta y_{i-1} \int_{I_i} f''(\xi_x)(x - x_{i-1})(x_i - x)(1 - \delta_i(x)) \, dx \\ &\quad + \Delta y_i \int_{I_i} f''(\xi_x)(x - x_{i-1})(x_i - x)\delta_i(x) \, dx \\ &= \frac{h_i^3}{12} (\Delta y_{i-1} f''(\eta_0) + \Delta y_i f''(\eta_1)), \end{aligned}$$

for some η_0 and η_1 in (x_{i-1}, x_i) .

Substituting previous results in equation (A.6),

$$\begin{aligned} \|f - \text{line}_i\|_{L_2(I_i)}^2 &= \frac{h_i}{3} ((\Delta y_{i-1})^2 + (\Delta y_i)^2 + \Delta y_{i-1} \Delta y_i) \\ &\quad + \frac{h_i^3}{12} (\Delta y_{i-1} f''(\eta_0) + \Delta y_i f''(\eta_1)) + \frac{h_i^5}{120} (f''(\eta))^2. \end{aligned}$$

We may now find the line segment that minimizes the distance to f by taking partial derivatives with respect to Δy_{i-1} and Δy_i , setting them to zero and solving the corresponding system of equations. Indeed, the previous error is quadratic in Δy_{i-1} and Δy_i , and attains its minimum at

$$\begin{cases} \Delta y_{i-1} &= (f''(\eta_1) - 2f''(\eta_0))h_i^2/12, \\ \Delta y_i &= (f''(\eta_0) - 2f''(\eta_1))h_i^2/12. \end{cases} \quad (\text{A.7})$$

The resulting minimum squared distance is given by equation (5.11).

A.3 Proof of theorem 5.4 (Asymptotic Analysis)

By the triangle inequality, the jump discontinuity at $x = x_i$ between two adjacent independently-optimized segments is

$$\begin{aligned} |\Delta y_i^- - \Delta y_i^+| &\leq \frac{h_i^2}{12} |f''(\eta_{0,i}) - 2f''(\eta_{1,i})| \\ &\quad + \frac{h_{i+1}^2}{12} |f''(\eta_{1,i+1}) - 2f''(\eta_{0,i+1})|, \end{aligned} \tag{A.8}$$

where $\Delta y_i^- = (f''(\eta_{0,i}) - 2f''(\eta_{1,i}))h_i^2/12$ and $\Delta y_i^+ = (f''(\eta_{1,i+1}) - 2f''(\eta_{0,i+1}))h_{i+1}^2/12$ are the offsets with respect to $f(x_i)$ of the optimized segments (equation (A.7)) at the left and right of $x = x_i$, respectively; $\eta_{0,j}$ and $\eta_{1,j}$ lie in the interval I_j . Since we are dealing with functions twice continuously differentiable in a closed interval, the absolute value terms in equation (A.8) are bounded, according to the extreme value theorem; therefore, if h_i and h_{i+1} decrease (finer partition T), the discontinuity jumps at the knots of the partition also decrease. In the limit, as $\sup h_i \rightarrow 0$, $|\Delta y_i^- - \Delta y_i^+| \rightarrow 0$, i.e., continuity is satisfied. Therefore the union of the independently-optimized segments $\text{line}_i \rightarrow P_T f$, which is the unique piecewise linear function satisfying both continuity ($\in V_T$) and minimization of the L_2 error (equation (5.7)). Consequently, if N is large we may approximate $\|f - P_T f\|_{L_2(I_i)}^2 \approx \min \|f - \text{line}_i\|_{L_2(I_i)}^2$; moreover, if I_i is sufficiently small so that f'' is approximately constant within it, f''_{I_i} , then we use Corollary 5.3 to get equation (5.13).

Bibliography

- [1] “PwC global entertainment and media outlook: 2015–2019,” PricewaterhouseCoopers, Tech. Rep., 2015.
- [2] “Top 500 supercomputer sites,” June 2015. [Online]. Available: <http://www.top500.org/lists/2015/06>
- [3] T. Akenine-Möller and B. Johnsson, “Performance per what?” *Journal of Computer Graphics Techniques (JCGT)*, vol. 1, no. 1, pp. 37–41, 2012.
- [4] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the Spring Joint Computer Conference*. ACM, April 1967, pp. 483–485.
- [5] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, “A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking,” *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002.
- [6] M. Auguin and G. Giraudon, “Image processing on a SIMD/SPMD architecture: OPSILA,” in *International Conference on Pattern Recognition*, vol. 1, Nov 1988, pp. 430–433.
- [7] S.-N. Ba, K. Waheed, and G. Zhou, “Efficient spacing scheme for a linearly interpolated lookup table predistorter,” in *IEEE International Symposium on Circuits and Systems*, 2008, pp. 1512–1515.
- [8] H. C. Baker, Jr. and C. Hewitt, “The incremental garbage collection of processes,” in *Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: ACM, 1977, pp. 55–59.
- [9] O. Barnich and M. Van Droogenbroeck, “ViBe: a powerful random technique to estimate the background in video sequences,” in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2009, pp. 945–948.
- [10] F. Belkhouche, “Trajectory-based optimal linearization for nonlinear autonomous vector fields,” *IEEE Transactions on Circuits and Systems—Part I: Regular Papers*, vol. 52, no. 1, pp. 127–138, 2005.
- [11] R. Bellman and R. Roth, “Curve fitting by segmented straight lines,” *Journal of the American Statistical Association*, vol. 64, no. 327, pp. 1079–1084, 1969.
- [12] D. Berjón, S. Arnaldo, and F. Morán, “A parallel implementation of 3D Zernike moment analysis,” in *Proceedings of SPIE*, vol. 7872, January 2011, pp. 787 209–1–787 209–7.

-
- [13] D. Berjón, C. Cuevas, F. Morán, and N. García, “Moving object detection strategy for augmented-reality applications in a GPGPU by using CUDA,” in *IEEE International Conference on Consumer Electronics*, January 2012, pp. 319–320.
 - [14] D. Berjón, C. Cuevas, F. Morán, and N. García, “GPU-based implementation of an optimized nonparametric background modeling for real-time moving object detection,” *IEEE Transactions on Consumer Electronics*, vol. 59, no. 2, pp. 361–369, 2013.
 - [15] D. Berjón, C. Cuevas, F. Morán, and N. García, “Region-based moving object detection using spatially conditioned nonparametric models in a GPU,” in *IEEE International Conference on Consumer Electronics*, January 2014, pp. 359–360.
 - [16] D. Berjón, G. Gallego, C. Cuevas, F. Morán, and N. García, “Optimal piecewise linear function approximation for GPU-based applications,” *IEEE Transactions on Cybernetics*, vol. x, no. x, pp. xx–yy, x 2016, (to appear).
 - [17] D. Berjón, A. Hornung, F. Morán, and A. Smolic, “Evaluation of backward mapping DIBR for FVV applications,” in *IEEE International Conference on Multimedia and Expo*, July 2011, pp. 1–6.
 - [18] M. Bicego, M. Cristani, and V. Murino, “Unsupervised scene analysis: A hidden Markov model approach,” *Computer Vision and Image Understanding*, vol. 102, no. 1, pp. 22 – 41, 2006.
 - [19] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.
 - [20] M. Bordallo López, “Designing for energy-efficient vision-based interactivity on mobile devices,” Ph.D. dissertation, University of Oulu, 2014.
 - [21] M. Bordallo López, J. Hannuksela, O. Silvén, and M. Vehviläinen, “Interactive multi-frame reconstruction for mobile devices,” *Multimedia Tools and Applications*, vol. 69, no. 1, pp. 31–51, March 2012.
 - [22] S. Borkar and A. A. Chien, “The future of microprocessors,” *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, May 2011.
 - [23] A. G. Bors and N. Nasios, “Kernel bandwidth estimation for nonparametric modeling,” *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics*, vol. 39, no. 6, pp. 1543–1555, 2009.
 - [24] T. Bouwmans, “Traditional and recent approaches in background modeling for foreground detection: An overview,” *Computer Science Review*, vol. 11, pp. 31–66, 2014.
 - [25] T. Bouwmans, F. El Baf, and B. Vachon, “Background modeling using mixture of gaussians for foreground detection—a survey,” *Recent Patents on Computer Science*, vol. 1, no. 3, pp. 219–237, 2008.
 - [26] I. Bronshtein, K. Semendyayev, G. Musiol, and H. Mühlig, *Handbook of Mathematics*, 6th ed. Springer, 2015.
 - [27] P. Brox, J. Castro-Ramírez, M. Martínez-Rodríguez, E. Tena, C. Jiménez, I. Baturone, and A. Acosta, “A programmable and configurable ASIC to generate piecewise-affine functions defined over general partitions,” *IEEE Transactions on Circuits and Systems—Part I: Regular Papers*, vol. 60, no. 12, pp. 3182–3194, 2013.

- [28] S. Brutzer, B. Höferlin, and G. Heidemann, “Evaluation of background subtraction techniques for video surveillance,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2011, pp. 1937–1944.
- [29] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: Stream computing on graphics hardware,” *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 777–786, Aug. 2004.
- [30] P. Carballera, F. Morán, and J. Cabrera, “View rectification for FTV ref. SW depth estimation improvement,” ISO/IEC JTC1/SC29/WG11, Archamps, France, contribution M15281, Apr. 2008.
- [31] J. Carranza, C. Theobalt, M. A. Magnor, and H. Seidel, “Free-viewpoint video of human actors,” *ACM Transactions on Graphics*, vol. 22, pp. 569–577, July 2003.
- [32] J. Cavers, “Optimum table spacing in predistorting amplifier linearizers,” *IEEE Transactions on Vehicular Technology*, vol. 48, no. 5, pp. 1699–1705, 1999.
- [33] H. J. Chang, H. Jeong, and J. Y. Choi, “Active attentional sampling for speed-up of background subtraction,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 2088–2095.
- [34] E. W. Cheney and D. R. Kincaid, “Errors in polynomial interpolation,” in *Numerical Mathematics and Computing*, 7th ed. Cengage Learning, 2012, ch. 4.2, p. 181.
- [35] S.-C. S. Cheung and C. Kamath, “Robust techniques for background subtraction in urban traffic video,” in *Proceedings of SPIE 5308 – Visual Communications and Image Processing*, vol. 5308, 2004, pp. 881–892.
- [36] D. Cireşan, U. Meier, J. Masci, and J. Schmidhuber, “Multi-column deep neural network for traffic sign classification,” *Neural Networks*, vol. 32, pp. 333 – 338, August 2012.
- [37] J. H. Clark, “The Geometry Engine: A VLSI geometry system for graphics,” *ACM Annual Conference on Computer Graphics and Interactive Techniques*, vol. 16, no. 3, pp. 127–133, Jul. 1982.
- [38] S. Collange, “GPU programming,” in *Graduate course at Universidade Federale de Minas Gerais*, October 2015.
- [39] M. E. Conway, “A multiprocessor system design,” in *Fall Joint Computer Conference*, ser. AFIPS ’63 (Fall). New York, NY, USA: ACM, 1963, pp. 139–146.
- [40] M. Cox, P. Harris, and P. Kenward, “Fixed- and free-knot univariate least-square data approximation by polynomial splines,” in *International Symposium on Algorithms for Approximation*, 2001, pp. 330–345.
- [41] R. Cucchiara, C. Grana, M. Piccardi, and A. Prati, “Detecting moving objects, ghosts, and shadows in video streams,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 10, pp. 1337–1342, October 2003.
- [42] C. Cuevas, D. Berjón, F. Morán, and N. García, “Moving object detection for real-time augmented reality applications in a GPGPU,” *IEEE Transactions on Consumer Electronics*, vol. 58, no. 1, pp. 117–125, 2012.

-
- [43] C. Cuevas, D. Berjón, F. Morán, and N. García, “GPGPU implementation of an improved nonparametric background modeling for moving object detection strategies,” in *IEEE International Conference on Consumer Electronics*, January 2013, pp. 23–24.
 - [44] C. Cuevas and N. García, “Improved background modeling for real-time spatio-temporal non-parametric moving object detection strategies,” *Image and Vision Computing*, vol. 31, no. 9, pp. 616–630, 2013.
 - [45] D. Culibrk, O. Marques, D. Socek, H. Kalva, and B. Furht, “Neural network approach to background modeling for video object segmentation,” *IEEE Transactions on Neural Networks*, vol. 18, no. 6, pp. 1614–1627, 2007.
 - [46] C. de Boor, “Piecewise linear approximation,” in *A Practical Guide to Splines*. Springer, 2001, ch. 3, pp. 31–37.
 - [47] M. Doggett, “Texture caches,” *IEEE Micro*, vol. 32, no. 3, pp. 136–141, May/June 2012.
 - [48] R. Douc and O. Cappé, “Comparison of resampling schemes for particle filtering,” in *International Symposium on Image and Signal Processing and Analysis*, 2005, pp. 64–69.
 - [49] B. Duan, W. Wang, X. Li, C. Zhang, P. Zhang, and N. Sun, “Floating-point mixed-radix FFT core generation for FPGA and comparison with GPU and CPU,” in *International Conference on Field-Programmable Technology*, Dec 2011, pp. 1–6.
 - [50] A. Elgammal, R. Duraiswami, and L. S. Davis, “Efficient non-parametric adaptive color modeling using fast gauss transform,” in *IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, 2001, pp. II–563.
 - [51] A. Elgammal, R. Duraiswami, D. Harwood, and L. S. Davis, “Background and foreground modeling using nonparametric kernel density estimation for visual surveillance,” *Proceedings of the IEEE*, vol. 90, no. 7, pp. 1151–1163, 2002.
 - [52] A. Elgammal, D. Harwood, and L. Davis, “Non-parametric model for background subtraction,” in *European Conference on Computer Vision*, ser. Lecture Notes in Computer Science, D. Vernon, Ed. Springer Berlin Heidelberg, 2000, vol. 1843, pp. 751–767.
 - [53] S. Y. Elhabian, K. M. El-Sayed, and S. H. Ahmed, “Moving object detection in spatial domain using background removal techniques — state-of-art,” *Recent Patents on Computer Science*, vol. 1, no. 1, pp. 32–54, 2008.
 - [54] K. Eriksson, D. Estep, and C. Johnson, “Piecewise linear approximation,” in *Applied Mathematics: Body and Soul*. Springer, 2004, vol. 2, no. 978-3-540-00889-7, ch. 52, pp. 743–754.
 - [55] R. H. Evangelio and T. Sikora, “Complementary background models for the detection of static and moving objects in crowded environments,” in *IEEE International Conference on Advanced Video- and Signal-Based Surveillance*. IEEE, 2011, pp. 71–76.
 - [56] C. Fehn, “Depth-image-based rendering (DIBR), compression, and transmission for a new approach on 3D-TV,” in *Proceedings of SPIE*, vol. 5291. Spie, 2004, pp. 93–104.
 - [57] L. J. Flynn, “Intel halts development of 2 new microprocessors,” *The New York Times*, 8th May 2004.
 - [58] M. Flynn, “Some computer organizations and their effectiveness,” *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sept 1972.

- [59] C. Frenzen, T. Sasao, and J. T. Butler, “On the number of segments needed in a piecewise linear approximation,” *Journal of Computational and Applied Mathematics*, vol. 234, no. 2, pp. 437–446, 2010.
- [60] D. P. Friedman and D. S. Wise, “Aspects of applicative programming for parallel processing,” *IEEE Transactions on Computers*, vol. C-27, no. 4, pp. 289–296, April 1978.
- [61] G. Gallego, D. Berjón, and N. García, “Optimal Polygonal L_1 Linearization and Fast Interpolation of Nonlinear Systems,” *IEEE Transactions on Circuits and Systems—Part I: Regular Papers*, vol. 61, no. 11, pp. 3225–3234, Nov 2014.
- [62] G. Gallego, C. Cuevas, R. Mohedano, and N. García, “On the Mahalanobis distance classification criterion for multidimensional normal distributions,” *IEEE Transactions on Signal Processing*, vol. 61, no. 17, pp. 4387–4396, 2013.
- [63] J. Gallego, M. Pardàs, and J.-L. Landabaso, “Segmentation and tracking of static and moving objects in video surveillance scenarios,” in *IEEE International Conference on Image Processing*, 2008, pp. 2716–2719.
- [64] S. Ghosh, A. Ray, D. Yadav, and B. M. Karan, “A genetic algorithm based clustering approach for piecewise linearization of nonlinear functions,” in *International Conference on Devices and Communications*, 2011, pp. 1–4.
- [65] C. Gregg and K. Hazelwood, “Where is the data? why you cannot debate CPU vs. GPU performance without the answer,” in *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011, pp. 134–144.
- [66] M. Guillaumin, T. Mensink, J. Verbeek, and C. Schmid, “Face recognition from caption-based supervision,” *International Journal of Computer Vision*, vol. 96, no. 1, pp. 64–82, 2012.
- [67] J. L. Gustafson, “Reevaluating Amdahl’s Law,” *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.
- [68] —, *The End of Error: Unum Computing*. CRC Press, 2015.
- [69] K. M. Gutttag, T. M. Albers, M. D. Asal, and K. G. Rose, “The TMS34010: an embedded microprocessor,” *IEEE Micro*, vol. 8, no. 3, pp. 39–52, June 1988.
- [70] T. S. Haines and T. Xiang, “Background subtraction with Dirichlet process mixture models,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 4, pp. 670–683, 2014.
- [71] P. B. Hansen, “The programming language Concurrent Pascal,” *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 199–207, June 1975.
- [72] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*, 2nd ed. Cambridge university press, 2003.
- [73] H. Hassanpour, M. Sedighi, and A. R. Manashty, “Video frame’s background modeling: Reviewing the techniques,” *Journal of Signal and Information Processing*, vol. 2, no. 2, pp. 72–78, 2011.

-
- [74] T. Hatanaka, K. Uosaki, and M. Koga, “Evolutionary computation approach to Wiener model identification,” in *Congress on Evolutionary Computation*, vol. 1, 2002, pp. 914–919.
- [75] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., 2011.
- [76] C. A. R. Hoare, “Monitors: An operating system structuring concept,” *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974.
- [77] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver, “Fast computation of generalized Voronoi diagrams using graphics hardware,” in *ACM Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286.
- [78] M. Hofmann, P. Tiefenbacher, and G. Rigoll, “Background segmentation with feedback: The pixel-based adaptive segmenter,” in *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, June 2012, pp. 38–43.
- [79] A. Hornung and L. Kobbelt, “Interactive pixel-accurate free viewpoint rendering from images with silhouette aware sampling,” *Computer Graphics Forum*, vol. 28, no. 8, pp. 2090–2103, 2009.
- [80] S. Hu, D. S. Poskitt, and X. Zhang, “Bayesian adaptive bandwidth kernel density estimation of irregular multivariate distributions,” *Computational Statistics & Data Analysis*, vol. 56, no. 3, pp. 732–740, 2012.
- [81] *1788-2015 - IEEE Standard for Interval Arithmetic*, IEEE Std.
- [82] M. Isard and A. Blake, “Condensation—conditional density propagation for visual tracking,” *International Journal of Computer Vision*, vol. 29, no. 1, pp. 5–28, 1998.
- [83] M. C. Jones, J. S. Marron, and S. J. Sheather, “A brief survey of bandwidth selection for density estimation,” *Journal of the American Statistical Association*, vol. 91, no. 433, pp. 401–407, 1996.
- [84] P. Julian, A. Desages, and O. Agamennoni, “High-level canonical piecewise linear representation using a simplicial partition,” *IEEE Transactions on Circuits and Systems—Part I: Fundamental Theory and Applications*, vol. 46, no. 4, pp. 463–480, 1999.
- [85] P. Julian, A. Desages, and B. D’Amico, “Orthonormal high-level canonical PWL functions with applications to model reduction,” *IEEE Transactions on Circuits and Systems—Part I: Fundamental Theory and Applications*, vol. 47, no. 5, pp. 702–712, 2000.
- [86] H. Kim, R. Sakamoto, I. Kitahara, T. Toriyama, and K. Kogure, “Background subtraction using generalised Gaussian family model,” *Electronics Letters*, vol. 44, pp. 189–190(1), January 2008.
- [87] K. Kim, T. H. Chalidabhongse, D. Harwood, and L. Davis, “Background modeling and subtraction by codebook construction,” in *IEEE International Conference on Image Processing*, vol. 5, 2004, pp. 3061–3064.
- [88] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2012.

- [89] J. G. Koomey, S. Berard, M. Sanchez, and H. Wong, “Implications of historical trends in the electrical efficiency of computing,” *IEEE Annals of the History of Computing*, vol. 33, no. 3, pp. 46–54, 2011.
- [90] N. P. Kronenberg, H. M. Levy, and W. D. Strecker, “VAXcluster: A closely-coupled distributed system,” *ACM Transactions on Computer Systems*, vol. 4, no. 2, pp. 130–146, May 1986.
- [91] E. S. Larsen and D. McAllister, “Fast matrix multiplies using graphics hardware,” in *ACM/IEEE Conference on Supercomputing*, ser. SC ’01. New York, NY, USA: ACM, 2001, pp. 55–55.
- [92] B. Lee and M. Hedley, “Background estimation for video surveillance,” in *Image and Vision Computing New Zealand*, 2002, pp. 315–320.
- [93] A. Lefohn, M. Houston, J. Andersson, U. Assarsson, C. Everitt, K. Fatahalian, T. Foley, J. Hensley, P. Lalonde, and D. Luebke, “Beyond programmable shading (parts I and II),” in *ACM SIGGRAPH 2009 Courses*, ser. SIGGRAPH ’09. New York, NY, USA: ACM, 2009, pp. 7:1–7:312.
- [94] L. Li, W. Huang, I. Y. Gu, and Q. Tian, “Foreground object detection from videos containing complex background,” in *ACM International Conference on Multimedia*, 2003, pp. 2–10.
- [95] L. Li, W. Huang, I. Y.-H. Gu, and Q. Tian, “Statistical modeling of complex backgrounds for foreground object detection,” *IEEE Transactions on Image Processing*, vol. 13, no. 11, pp. 1459–1472, 2004.
- [96] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A unified graphics and computing architecture,” *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.
- [97] B. Lo and S. Velastin, “Automatic congestion detection system for underground platforms,” in *International Symposium on Intelligent Multimedia, Video and Speech Processing*, 2001, pp. 158–161.
- [98] R. Luque, E. Domínguez, E. Palomo, and J. Muñoz, “A neural network approach for video object segmentation in traffic surveillance,” in *International Conference on Image Analysis and Recognition*, ser. Lecture Notes in Computer Science, A. Campilho and M. Kamel, Eds. Springer Berlin Heidelberg, 2008, vol. 5112, pp. 151–158.
- [99] L. Maddalena and A. Petrosino, “A self-organizing approach to background subtraction for visual surveillance applications,” *IEEE Transactions on Image Processing*, vol. 17, no. 7, pp. 1168–1177, 2008.
- [100] —, “The SOBS algorithm: what are the limits?” in *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2012, pp. 21–26.
- [101] A. Marchand and T. Hennig-Thurau, “Value creation in the video game industry: Industry economics, consumer benefits, and research opportunities,” *Journal of Interactive Marketing*, vol. 27, no. 3, pp. 141–157, 2013.
- [102] W. Mark, “Post-rendering 3D image warping: Visibility, reconstruction, and performance for depth-image warping,” Ph.D. dissertation, University of North Carolina, 1999.

-
- [103] N. Martel-Brisson and A. Zaccarin, “Unsupervised approach for building non-parametric background and foreground models of scenes with significant foreground activity,” in *ACM Workshop on Vision Networks for Behavior Analysis*, ser. VNBA '08. New York, NY, USA: ACM, 2008, pp. 93–100.
- [104] R. Martínez, C. Cuevas, D. Berjón, and N. García, “Detection of static moving objects using multiple nonparametric background models,” in *IEEE International Symposium on Consumer Electronics*, June 2015, pp. 1–2.
- [105] W. Matusik, C. Buehler, and L. McMillan, “Polyhedral visual hulls for real-time rendering,” in *Eurographics Workshop on Rendering Techniques*. London, UK: Springer-Verlag, 2001, pp. 115–126.
- [106] A. J. W. Mayer, “The architecture of the Burroughs B5000: 20 years later and still ahead of the times?” *SIGARCH Computer Architecture News*, vol. 10, no. 4, pp. 3–10, Jun. 1982.
- [107] N. J. McFarlane and C. P. Schofield, “Segmentation and tracking of piglets in images,” *Machine Vision and Applications*, vol. 8, no. 3, pp. 187–193, 1995.
- [108] S. J. McKenna, S. Jabri, Z. Duric, A. Rosenfeld, and H. Wechsler, “Tracking groups of people,” *Computer Vision and Image Understanding*, vol. 80, no. 1, pp. 42–56, 2000.
- [109] L. McMillan, “An image-based approach to three-dimensional computer graphics,” Ph.D. dissertation, University of North Carolina, 1997.
- [110] P. Merkle, A. Smolic, K. Muller, and T. Wiegand, “Multi-view video plus depth representation and coding,” in *IEEE International Conference on Image Processing*, vol. 1, Sept 2007, pp. I–201–I–204.
- [111] L. Mihaylova, A. Y. Carmi, F. Septier, A. Gning, S. K. Pang, and S. Godsill, “Overview of bayesian sequential Monte Carlo methods for group and extended object tracking,” *Digital Signal Processing*, vol. 25, pp. 1–16, 2014.
- [112] D. Min, D. Kim, S. Yun, and K. Sohn, “2D/3D freeview video generation for 3DTV system,” *Signal Processing: Image Communication*, vol. 24, no. 1-2, pp. 31–48, Jan. 2009.
- [113] A. Mittal and N. Paragios, “Motion-based background subtraction using adaptive kernel density estimation,” in *IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, June 2004, pp. II–302–II–309 Vol.2.
- [114] D. Monniaux, “The pitfalls of verifying floating-point computations,” *ACM Transactions on Programming Languages and Systems*, vol. 30, no. 3, pp. 12:1–12:41, May 2008.
- [115] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, p. 114, April 1965.
- [116] Y. Mori, N. Fukushima, T. Fujii, and M. Tanimoto, “View generation with 3D warping using depth information for FTV,” *Signal Processing: Image Communication*, vol. 22, no. 2, pp. 229–232, May 2008.
- [117] K. Müller, P. Merkle, and T. Wiegand, “3-D video representation using depth maps,” *Proceedings of the IEEE*, vol. 99, no. 4, pp. 643–656, April 2011.

- [118] K. Müller, A. Smolic, K. Dix, P. Merkle, P. Kauff, and T. Wiegand, “View synthesis for advanced 3D video systems,” *EURASIP Journal on Image and Video Processing*, vol. 2008, pp. 1–12, 2008.
- [119] J. Nickolls and W. J. Dally, “The GPU computing era,” *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.
- [120] A. Nieto, D. L. Vilarino, and V. Brea Sánchez, *Machine Vision – Applications and Systems*. InTech, 2012, ch. Towards the Optimal Hardware Architecture for Computer Vision.
- [121] NVIDIA Corporation, “CUDA C programming guide,” NVIDIA Corporation, Tech. Rep., 2012.
- [122] K.-J. Oh, S. Yea, and Y.-S. Ho, “Hole filling method using depth based in-painting for view synthesis in free viewpoint television and 3-D video,” in *Picture Coding Symposium*, May 2009, pp. 1–4.
- [123] N. M. Oliver, B. Rosario, and A. P. Pentland, “A Bayesian computer vision system for modeling human interactions,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 831–843, 2000.
- [124] R. Pagés, D. Berjón, and F. Morán, “Automatic system for virtual human reconstruction with 3d mesh multi-texturing and facial enhancement,” *Signal Processing: Image Communication*, vol. 28, no. 9, pp. 1089–1099, October 2013.
- [125] R. Pagés, D. Berjón, F. Morán, and N. García, “Seamless, static multi-texturing of 3d meshes,” *Computer Graphics Forum*, vol. 34, no. 1, pp. 228–238, February 2015.
- [126] V. K. Pallipuram, N. Raut, X. Ren, M. C. Smith, and S. Naik, “A multi-node GPGPU implementation of non-linear anisotropic diffusion filter,” in *Symposium on Application Accelerators in High Performance Computing*, 2012, pp. 11–18.
- [127] N. Paragios and V. Ramesh, “A MRF-based approach for real-time subway monitoring,” in *IEEE Conference on Computer Vision and Pattern Recognition*, vol. 1, 2001, pp. I-1034–I-1040.
- [128] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. Kim, “Design and performance evaluation of image processing algorithms on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 91–104, Jan 2011.
- [129] E. Parzen, “On estimation of a probability density function and mode,” *Annals of Mathematical Statistics*, vol. 33, no. 3, pp. 1065–1076, 09 1962.
- [130] G. F. Pfister, *In Search of Clusters*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.
- [131] M. Piccardi, “Background subtraction techniques: a review,” in *IEEE International Conference on Systems, Man and Cybernetics*, vol. 4, 2004, pp. 3099–3104.
- [132] F. Porikli, Y. Ivanov, and T. Haga, “Robust abandoned object detection using dual foregrounds,” *EURASIP Journal on Advances in Signal Processing*, vol. 2008, p. 30, 2008.
- [133] M. Potmesil, “Generating octree models of 3D objects from their silhouettes in a sequence of images,” *Computer Vision, Graphics, and Image Processing*, vol. 40, no. 1, pp. 1–29, Oct. 1987.

-
- [134] B. Rister, G. Wang, M. Wu, and J. R. Cavallaro, "A fast and efficient SIFT detector using the mobile GPU," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 2674–2678.
- [135] J. Rittscher, J. Kato, S. Joga, and A. Blake, "A probabilistic background model for tracking," in *European Conference on Computer Vision*, ser. ECCV '00. Springer-Verlag, 2000, pp. 336–350.
- [136] R. Rojas, "How to make Zuse's Z3 a universal computer," *IEEE Annals of the History of Computing*, vol. 20, no. 3, pp. 51–54, Jul 1998.
- [137] M. Rosenblatt, "Remarks on some nonparametric estimates of a density function," *Annals of Mathematical Statistics*, vol. 27, no. 3, pp. 832–837, 09 1956.
- [138] R. Rossi, S. A. Tarim, S. Prestwich, and B. Hnich, "Piecewise linear approximations of the standard normal first order loss function," *arXiv:1307.1708 [math.OC]*, 2013.
- [139] M. Sehili, D. Istrate, B. Dorizzi, and J. Boudy, "Daily sound recognition using a combination of GMM and SVM for home automation," in *European Signal Processing Conference*, Bucharest, 2012, pp. 1673–1677.
- [140] J. Shade, S. Gortler, L. He, and R. Szeliski, "Layered depth images," in *ACM Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 231–242.
- [141] Y. Sheikh, O. Javed, and T. Kanade, "Background subtraction for freely moving cameras," in *IEEE International Conference on Computer Vision*, Sept 2009, pp. 1219–1225.
- [142] Y. Sheikh and M. Shah, "Bayesian modeling of dynamic scenes for object detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 11, pp. 1778–1792, 2005.
- [143] A. Shimada, H. Nagahara, and R.-i. Taniguchi, "Background modeling based on bidirectional analysis," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 1979–1986.
- [144] K. Shoemake, "Animating rotation with quaternion curves," *ACM Annual Conference on Computer Graphics and Interactive Techniques*, vol. 19, no. 3, pp. 245–254, Jul. 1985.
- [145] A. W. Smeulders, D. M. Chu, R. Cucchiara, S. Calderara, A. Dehghan, and M. Shah, "Visual tracking: an experimental survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 7, pp. 1442–1468, 2014.
- [146] A. Sobral and A. Vacavant, "A comprehensive review of background subtraction algorithms evaluated with synthetic and real videos," *Computer Vision and Image Understanding*, vol. 122, pp. 4–21, 2014.
- [147] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter, "The tradeoffs of fused memory hierarchies in heterogeneous computing architectures," in *Conference on Computing Frontiers*, ser. CF '12. New York, NY, USA: ACM, 2012, pp. 103–112.
- [148] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," in *IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, 1999, pp. 246–252.

- [149] G. Stockman and L. G. Shapiro, *Computer Vision*, 1st ed. Prentice Hall PTR, 2001.
- [150] M. Storace and O. De Feo, "Piecewise-linear approximation of nonlinear dynamical systems," *IEEE Transactions on Circuits and Systems—Part I: Regular Papers*, vol. 51, no. 4, pp. 830–842, 2004.
- [151] M. Sylwestrzak, D. Szlag, M. Szkulmowski, I. Gorczyńska, D. Bukowska, M. Wojtkowski, and P. Targowski, "Real time 3D structural and doppler OCT imaging on graphics processing units," in *Proceedings of SPIE*, no. 85710Y. International Society for Optics and Photonics, 2013.
- [152] M. Tanimoto, M. Tehrani, T. Fujii, and T. Yendo, "Free-viewpoint TV," *IEEE Signal Processing Magazine*, vol. 28, no. 1, pp. 67–76, Jan. 2011.
- [153] R. Tanjad and S. Wongsu, "Model structure selection strategy for Wiener model identification with piecewise linearisation," in *International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, 2011, pp. 553–556.
- [154] A. Tavakkoli, M. Nicolescu, G. Bebis, and M. Nicolescu, "Non-parametric statistical background modeling for efficient foreground region detection," *Machine Vision and Applications*, vol. 20, no. 6, pp. 395–409, 2009.
- [155] A. Telea, "An image inpainting technique based on the fast marching method," *Journal of Graphics Tools*, vol. 9, no. 1, pp. 23–34, 2004.
- [156] L. H. Thomas, "Elliptic problems in linear difference equations over a network," Watson Scientific Computing Laboratory, Columbia University, Tech. Rep., 1949.
- [157] C. J. Thompson, S. Hahn, and M. Oskin, "Using modern graphics architectures for general-purpose computing: A framework and analysis," in *Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 35. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 306–317.
- [158] D. Tian, P. Lai, P. Lopez, and C. Gomila, "View synthesis techniques for 3D video," in *Proceedings of SPIE*, vol. 7443, no. 609. Spie, 2009, pp. 74 430T–74 430T–11.
- [159] L. Torres Quevedo, "Ensayos sobre automática. su definición. extensión teórica de sus aplicaciones," *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales*, vol. XII, pp. 391–419, 1914.
- [160] A. Utasi and L. Czuni, "HMM-based unusual motion detection without tracking," in *International Conference on Pattern Recognition*, Dec 2008, pp. 1–4.
- [161] K. E. A. van de Sande, T. Gevers, and C. G. M. Snoek, "Empowering visual categorization with the GPU," *IEEE Transactions on Multimedia*, vol. 13, no. 1, pp. 60–70, Feb 2011.
- [162] M. P. Wand and M. C. Jones, *Kernel Smoothing*, ser. Monographs on statistics and applied probability. London: Chapman and Hall, 1995, no. 60.
- [163] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2010, pp. 235–246.

- [164] C. R. Wren, A. Azarbayejani, T. Darrell, and A. P. Pentland, "Pfinder: real-time tracking of the human body," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 7, pp. 780–785, Jul 1997.
- [165] Z. Xie and L. Guan, "Multimodal Information Fusion of Audio Emotion Recognition Based on Kernel Entropy Component Analysis," in *IEEE International Symposium on Multimedia*, Dec. 2012, pp. 1–8.
- [166] L. Yang, T. Yendo, M. P. Tehrani, T. Fujii, and M. Tanimoto, "Artifact reduction using reliability reasoning for image generation of FTV," *Journal of Visual Communication and Image Representation*, vol. 21, no. 5-6, pp. 542–560, Jul. 2010.
- [167] T.-Y. Yeh, D. T. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 2014, pp. 183–192.
- [168] X. Zhang and J. Yang, "Foreground segmentation based on selective foreground model," *Electronics Letters*, vol. 44, no. 14, pp. 851–852, July 2008.
- [169] C. L. Zitnick, S. B. Kang, M. Uyttendaele, S. Winder, and R. Szeliski, "High-quality video view interpolation using a layered representation," *ACM Annual Conference on Computer Graphics and Interactive Techniques*, vol. 1, no. 212, p. 600, 2004.
- [170] Z. Zivkovic and F. van der Heijden, "Efficient adaptive density estimation per image pixel for the task of background subtraction," *Pattern Recognition Letters*, vol. 27, no. 7, pp. 773–780, 2006.