About This Digital Edition

Original thesis **Academic Year**

Analysis and Design of High-Capacity Two-Dimensional Codes. Development of the Reader for Desktop and Mobile Environments. 2009/2010

Original academic context. This bilingual edition, version 1.0, presents Marco Querini's Italian-language Master's thesis, defended on July 23, 2010, in the Computer Engineering Degree Programme, Faculty of Engineering, University of Rome Tor Vergata.

Bibliographic correction. At the July 2010 defense, the first conference paper on the new symbology had not yet appeared; the bibliography therefore used the provisional title *High Capacity Colored QR Codes* and preliminary conference details. This edition gives the final publisher-registered citation, whose neutral title also avoids suggesting that the proposed color symbology was itself a QR Code: *High Capacity Colored Two Dimensional Codes*, IMCSIT 2010, pp. 709–716, doi:10.1109/IMCSIT.2010.5679869.

Trademark and scope. "QR Code" is a registered trademark of DENSO WAVE INCORPORATED in Japan and other countries. The chapter entitled *High-Capacity Color Quick Response Codes* (*Quick Response a Colori ad Alta Capacità*) examines how color might be introduced into the QR Code structure to increase data capacity further. The title uses "QR Code" solely to identify the subject of that academic inquiry; it does not imply that the chapter or the format—called HCC2D in publications subsequent to the thesis—is sponsored, endorsed, or affiliated with DENSO WAVE INCORPORATED.

The volume is arranged in two parts:

1. an English translation, presented first for international accessibility;
2. *Italiano — testo originale del 2010*, preserving the original Italian sequence and text, apart from the disclosed editorial treatment.

The original 2010 Italian thesis, conserved by the author and the University, is authoritative and prevails in the event of any discrepancy, ambiguity, or difference in interpretation. The translation follows its terminology and historical context without silently updating technical claims to reflect later standards or products.

In each language sequence, an editorial note records the prudential omission of the original Chapter 2 to avoid restating the detailed QR Code specification elements discussed there.

Figure 1.3 is a new vector table with independent schematics and published maximum-capacity facts. The standard-derived tables in Figures 3.1–3.4 are not reproduced: original conceptual diagrams retain only their relationships and calculation logic. Other figures and tables before Chapter 4 are author reconstructions from editable sources. Figures 5.1, 5.2, and 5.7 carry disclosed interventions; Figure 5.7 retains its experimental values, with neutral headings and an adjacent blue note recording the later HCC2D name. From Chapter 4 onward, the other figures remain historical evidence from the original research.

Editorial conventions. Dark-blue 2026 notes mark additions and revisions. The Italian prose retained in black follows the 2010 text, with one typographical correction; every omission or replacement is disclosed by an adjacent blue note.

Related resources:

- *Technical specification.* The later HCC2D format is documented in the public specification *HCC2D Code Specification*, version 0.9.0.
- *Official website.* hcc2d.com.

Bilingual digital edition 2026 — Version 1.0
Edizione digitale bilingue 2026 — Versione 1.0

I dedicate my thesis to those who stood by me throughout these years of study,
especially my parents and my sister.

Contents

About This Digital Edition **I**

Introduction **1**

1 Two-Dimensional Codes: Study and Analysis **5**
1.1 Barcodes . . . 5
1.1.1 A Simple Definition . . . 6
1.1.2 From Analog to Digital . . . 6
1.1.3 Evolution of Barcodes . . . 7
1.2 Two-Dimensional Codes . . . 7
1.2.1 Ambiguity of the Definition . . . 7
1.2.2 Types of Two-Dimensional Code . . . 8
1.2.3 Stacked 2D Barcodes: PDF417 and Codablock . . . 9
1.2.4 Two-Dimensional Matrix Codes . . . 10
1.2.5 Color 2D Codes: Microsoft HCCB . . . 12

2 The Quick Response Code (ISO/IEC 18004) **14**

3 High-Capacity Color Quick Response Codes **15**
3.1 Project Objectives . . . 16
3.1.1 Project Scope . . . 16
3.2 Open Problems . . . 16
3.2.1 Observations on Function Patterns . . . 17
3.2.2 Observations on the Encoding Area . . . 17
3.2.3 Principal Problems and Challenges . . . 18
3.3 Proposed Design Specifications . . . 19
3.3.1 Extending the Quick Response Tables . . . 20
3.3.2 Managing the Introduction of Color into Quick Response Codes 25
3.3.3 Reworking the Concept of Data Masking . . . 27
3.3.4 Defining a Chromatic-Distance Metric . . . 30

4 Design and Implementation of the Generator and Reader **31**
4.1 Reference Projects 31
4.2 Introduction to Android OS 32
4.3 Impact of Android OS 33
4.3.1 Code-Portability Limitations 33
4.3.2 An Emulator Inadequate for the Project 33
4.3.3 Camera Use: Project Risks and Development Considerations . 33
4.4 Generator Implementation 35
4.4.1 Source Structure 35
4.5 Reader Implementation 38
4.5.1 Project Structure 38
4.5.2 Architecture-Independent Core Packages 39
4.5.3 Platform-Dependent Packages 40
4.5.4 Backward-Compatibility Characteristics 41
4.5.5 Operational Summary of the Added Software Modules 41

5 Experimentation and Analysis of Results **43**
5.1 Operational Examples 43
5.2 Analysis of the Added Overhead 47
5.3 Code Reliability 48

6 Conclusions and Future Work **50**

List of Figures **51**

Bibliography **53**

Italiano — testo originale del 2010 **54**

Introduction

Today, barcodes are primarily used as tools for identifying commercial products. Since they appear on nearly every item offered for sale, their role in shops is widely known; nevertheless, they also show potential for new uses. They are objects made up of contrasting graphical elements that express an optical, machine-readable representation of data through the use of optical readers. They are tools capable of turning even printed paper into a digital medium. Reading a barcode is therefore an automatic information-retrieval process offering high performance in terms of reliability, accuracy, and decoding speed.

The need to innovate barcodes arose from their single yet critical flaw: an extremely limited storage capacity, resulting from an information density of only a few characters per square inch. There is no doubt that the conventional barcode is ideally suited to its most frequent use, which generally consists of representing a simple identifier for a given product. Its one-dimensional structure makes reading extremely fast and reliable. To exploit the latent potential of these objects and support new use cases, however, it became necessary to introduce a two-dimensional structure. One scenario enabled by the new technology is the ability to represent directly, instead of the product identifier alone, all relevant information associated with it (manufacturer, model, any technical specifications, and so forth). This led to the emergence of so-called 2D codes, which use both dimensions to encode data and have either a bar-based or matrix-based structure, although by an abuse of terminology both types are commonly called barcodes.

Despite the high capacities achieved with 2D technology, these symbologies still contain a constraint that limits their performance: because the contrasting elements can have only two values, white or black, an individual cell of a matrix code can never carry more than one bit of information.

This thesis concerns the innovation of two-dimensional codes by increasing information density through the introduction of color, thereby allowing each individual cell (or module) of the code to represent graphically more than one bit of information. In particular, the purpose of increasing code capacity is not so much to replace conventional barcodes in established applications with a new type of code, but rather to produce symbols that can support new functions by means of a greater information density.

In this respect, the most recent impetus for barcode innovation has come from

the widespread adoption of mobile devices such as modern smartphones. Once the appropriate application is installed, these devices can operate as optical readers by using their integrated cameras. This suggests a new possibility: a future scenario in which anyone carrying a mobile phone also has a potential 2D-code reader. The reference scenario consequently changes from use cases involving a limited group of users equipped with special-purpose hardware, typically for professional purposes, to entirely new use cases involving the mass market.

It is therefore clear that the ability to represent more data in the same area, which entails producing symbols of greater density, is one of the principal challenges for the next generation of barcodes.

The contribution of this work is the development of a new high-capacity symbol technology whose structural basis is drawn from Quick Response codes (ISO/IEC 18004). Their desirable characteristics include rapid reading and reliability, the latter achieved through excellent handling of distortion, misalignment, and error correction. Indeed, reading codes presents decidedly nontrivial image-processing and computer-vision challenges. A scanned image may depict a symbol very differently from the original: no assumptions can be made about the orientation in which the reader perceives the code; a square symbol, for example, may be deformed into a rectangle or an arbitrary quadrilateral by a shot that projects it along one dimension; and correct operation must also be guaranteed when codes are partially damaged or ambient lighting is less than optimal.

Within this context, the work addresses further challenges focused on handling the introduction of color, which at a minimum raises the problems of detecting and correcting the resulting chromatic distortions. In addition to the image-processing issues, a complex object such as a Quick Response code must be remodeled in such a way that redesigning its specifications does not undermine symbol reliability; the trade-off between increased capacity and code readability must be respected. Once these conceptual obstacles have been overcome, the implementation problems of a generator and reader for the proposed code symbology are addressed. Acceptable performance must also be obtained when a mobile device produces low-quality images whose chromatic information is subsampled, with each pixel represented on average by only 12 bits, eight of which are devoted solely to luminance.

Problems associated with printing on ordinary paper are also considered, as are the practical difficulties caused by the absence of color-printing facilities. The proposed symbology therefore does not constrain the actual color scheme, which remains customizable according to the requirements at hand. Practical solutions include codes using four shades of gray for black-and-white printers, or primary colors such as cyan and magenta when printers reproduce composite colors imprecisely. Performance is nevertheless affected by these choices, and not every possible color scheme produces acceptable results.

The reader developed for desktop systems is expected to outperform its mobile counterpart because of the better quality of desktop scans. Nonetheless, the latent

near-future potential of smartphones is apparent, as the optical hardware supplied with them improves year after year. It is easy to imagine that, in the years ahead, cameras with flash, autofocus, and high resolution will be present on most consumer devices. Although such devices are already fairly widespread, they do not yet cover the entire market: a non-negligible share of people still use earlier-generation phones, which will gradually disappear.

The originality of this work does not, however, lie in using smartphones to read codes, since barcode readers are already available both for devices running the Android operating system and for iPhones. Rather, recognizing the potential of these devices led to the idea of redesigning codes such as Quick Response. Although still deservedly current, when QR codes were conceived they could not have taken into account either the later emergence of devices such as smartphones or the widespread availability of low-cost color printers at that time[1].

The idea of producing high-capacity color codes is therefore highly topical, given that Microsoft itself has recently developed a technology called "High Capacity Color Barcode," which likewise introduces color into symbols. Microsoft's solution, however, has the disadvantage of being proprietary, and the structure designed for its codes is no better than those adopted by symbologies such as Quick Response or Data Matrix. This is why remodeling an already mature technology such as QR to support a chromatic component can potentially yield a significant result.

Structure of the Thesis

Chapter 1 analyzes the state of the art in two-dimensional bar-based and matrix-based codes, following an introductory discussion of conventional one-dimensional barcodes intended to introduce the subject. Its ultimate aim is to provide an understanding of the technologies currently in use, which serves as the starting point for deciding which structure it makes sense to remodel in order to introduce a color symbology. Numerous varieties of two-dimensional code exist. Their characteristics are sometimes similar, but their structures differ and may or may not offer advantages in particular respects. Quick Response was selected for reworking because of its high capacity, reliability, and decoding speed, although Data Matrix technology is by no means inferior. Choosing which code to rework and redesign into a high-capacity symbol that exploits chromatic information necessarily required consideration of the qualities of the various code types.

Chapter 2 therefore examines the structural characteristics of a Quick Response symbol in detail. Before reasoning about which specifications should be remodeled and how, it is clearly necessary to establish the functions provided by every individual internal component.

Chapter 3 defines the new specifications, carefully selecting what should be inherited and what should be reworked while above all avoiding the substantial risk of

[1] The period in question was the 1990s, when even the first generation of mobile phones had barely appeared.

producing a code with internal inconsistencies that degrade performance or compromise reliability. It consequently addresses image-processing problems that account for color management, manipulates the tables contained in the standard specifications, changes the way data are distributed over the symbol surface, reinvents the Data Masking procedure itself, and seeks to define a chromatic-distance metric.

Chapter 4 describes the implementation of a generator and reader for the new code symbology, demonstrating that the revised specifications remain consistent because they form the basis of symbols that experiments found to be readable in practice. The applications produced operate in both desktop and mobile environments; in the latter case they run on devices from various hardware manufacturers, including HTC, Samsung, LG, and Motorola, that use the Android operating system.

The fifth and final chapter presents several examples of high-capacity Quick Response code generation, together with operational tests of the reader on mobile devices such as the HTC Tattoo and HTC Magic. It also provides quantitative measurements of the computational overhead introduced by the additional software modules required to handle chromatic issues. Experiments were conducted on dozens of instances of the new code symbology for this purpose.

Finally, the work seeks to determine the greatest achievable information density at which a code produced and printed on ordinary paper by a nonprofessional printer remains readable. Printing tests generated codes of a predetermined physical size in order to establish the total capacity that a code belonging to the new symbology can actually provide in practice.

Chapter 1

Two-Dimensional Codes: Study and Analysis

This chapter surveys the state of the art in two-dimensional codes, with particular attention to the Quick Response ("QR") standard. To understand fully the design of High-Capacity Quick Response codes—the central objective of this work, together with the implementation of the corresponding optical reader—it is first necessary to establish the current state of the technology. For completeness, conventional barcodes are introduced initially. Although they are not two-dimensional, they are the origin of the technology considered here; because they are also common in the modern world, they provide a convenient introduction for readers who are not specialists. A section on the currently predominant classes and types of two-dimensional code follows, leading to the ISO/IEC 18004 standard for Quick Response codes discussed in subsequent chapters.

1.1 Barcodes

Today, everyone has some idea of what a barcode is, because its use has become an integral part of everyday life. Not everyone could explain the optical reading process involved, of course, but it is impossible never to have seen a cashier trying to scan the code on a product being purchased. This is only one of many use cases, although probably among the most common. Less obvious is the use of barcodes to identify trains traveling along railway lines. Barcode technology is pervasive: it tends to spread everywhere, and its use has become so ordinary that we scarcely notice it before our eyes. Barcodes appear on countless documents, such as invoices and tickets of various kinds, without attracting our attention.

What, then, does the use of barcodes provide? Functionally, it enables the au-

tomation of processes such as identifying the objects to which codes are attached, as already noted.

1.1.1 A Simple Definition

The simplest way to define a barcode is as a set of contrasting graphical elements arranged so that they can be read by an optical sensor. The scanned image of the code as perceived by the sensor—and therefore not necessarily identical to the original code because distortions may be present—must be decoded to recover the information it contains.

Figure 1.1: An example of an EAN-8 barcode.
Editorial note (2026): This EAN-8 example was independently generated by the author for the digital edition and does not reproduce the 2010 raster sample.

More precisely, a barcode is an optical representation of data that can be interpreted by a machine (a machine-readable format) using an optical medium. A sheet of paper bearing a barcode becomes a digital medium, conceptually not very different from a compact disc; what changes is the information density and hence the storage capacity, which is very low in barcodes.

1.1.2 From Analog to Digital

It is interesting to observe how the reading of a generic code reproduces the concept of analog-to-digital conversion in the processing of scans. For example, low or high luminance perceived by an optical sensor in a particular area of a code could be translated into binary language by evaluating it against a threshold (low luminance = 1 and high luminance = 0, or vice versa). This concept will be developed more extensively for two-dimensional codes. By contrast, because conventional barcodes use a one-dimensional symbology, they encode data only through the widths of and distances between parallel lines. The underlying principle remains the extraction of digital data from input characterized by analog values: the widths and distances perceived by the sensor cannot be identical to the corresponding quantities in the original code and must be evaluated in an attempt to reconstruct the correct values and, from them, the stored information. It should also be noted that merely identifying objects of a particular shape in a scanned image is a nontrivial problem. Although this subject is revisited later, it is worth emphasizing from the outset that the code image perceived by the optical reader is not free from distortions of various kinds.

1.1.3 Evolution of Barcodes

The weak point of barcodes is probably their low information density, which permits the storage of only a few characters in an area typically measuring several square centimeters. Consider the height of the code in Figure 1.1: increasing it would not increase the amount of information the code can contain, because only the horizontal component encodes data through the widths of and distances between parallel lines. The vertical component is kept tall instead of being reduced, for example, to a single millimeter solely to facilitate reading. Even a hurried and imprecise user can retrieve the data with an optical reader if the bars are sufficiently tall, whereas a code with one-millimeter bars requires a very steady and accurate hand.

The excellent reliability of barcodes led to their widespread adoption, but the severe capacity constraint greatly limits possible new uses (the complete code in Figure 1.1, for example, encodes only eight characters). This created a need to design codes structurally different from barcodes and representing their evolution. These codes abandon the one-dimensional structure and make full use of both dimensions to encode data, while retaining sufficient reliability.

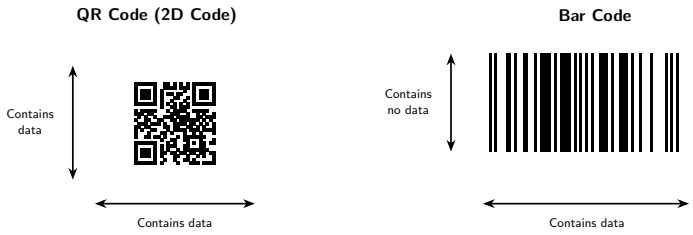

Figure 1.2: Comparison of a 1D code (barcode) and a 2D code (here a QR Code).
Editorial note (2026): This comparison is an original schematic independently created by the author for the digital edition; it does not reproduce the graphical samples in the 2010 figure.

1.2 Two-Dimensional Codes

This section discusses the principal types of two-dimensional code before examining one specific type, Quick Response, in depth. Understanding the standard that defines QR Codes is strictly necessary as a basis for understanding the modifications and extensions designed and implemented to produce the high-capacity Quick Response codes that are the objective of this thesis.

1.2.1 Ambiguity of the Definition

Two-dimensional codes exploit both dimensions to encode data and thereby maximize capacity. Some texts improperly call them two-dimensional barcodes even when they

contain no bars. This discussion seeks to avoid that confusion where possible. To clarify the terminology, one-dimensional codes are barcodes, while two-dimensional codes may be either bar-based or "matrix-based." Two-dimensional bar-based codes are also known as "stacked barcodes." They consist essentially of several juxtaposed one-dimensional barcodes and can therefore be decoded by conventional readers. Matrix codes, on the other hand, abandon any attempt to reproduce the earlier technology and adopt a regular checkerboard structure.

1.2.2 Types of Two-Dimensional Code

The varieties of two-dimensional code produced in recent years may differ significantly in performance, but all can encode optical representations of far more data than one-dimensional barcodes: capacities of several kilobytes are possible, compared with only a few bytes in one-dimensional codes. Within the two principal categories, "stacked" and "matrix," a wide variety of codes exists. PDF417, Codablock, and SuperCode belong to the first type, while MaxiCode, Data Matrix, and Quick Response are examples of matrix codes.

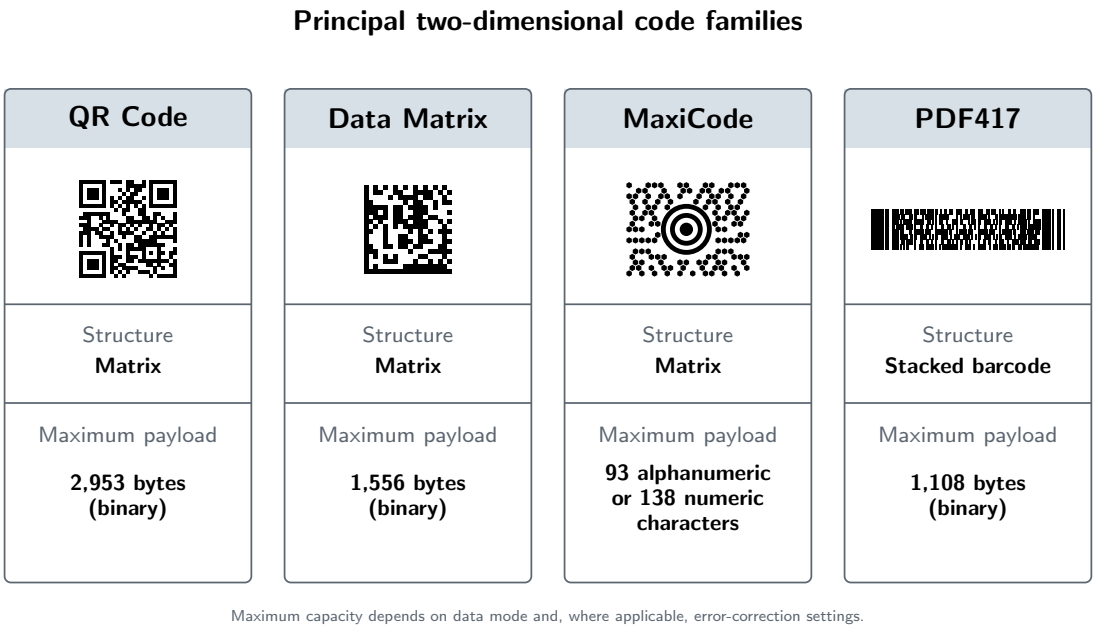

Figure 1.3: Comparison of the principal two-dimensional code families and their maximum capacities.
Editorial note (2026): This new vector table and its small symbol schematics were independently created by the author for the digital edition; they do not reproduce images from the standards. The QR Code, Data Matrix, and PDF417 miniatures encode "HCC2D digital edition"; MaxiCode is a structural schematic. The stated values are published maximum capacities. Because MaxiCode has no directly comparable binary limit, its maximum alphanumeric and numeric capacities are shown. Technical references: ISO/IEC 18004, ISO/IEC 16022, ISO/IEC 16023, and ISO/IEC 15438.

All the codes discussed so far share a structure made from contrasting black and white components. If information bits are associated with code areas on the basis

of chromatic content or luminance alone, every sampled value carries exactly one bit of information (white/light = 0 and black/dark = 1, or vice versa). Moreover, these codes are not recent and met with limited success for years. The era of smartphones, however, combines excellent integrated cameras with base software increasingly similar to the operating systems of personal computers. This has created strong momentum for using mobile devices as optical readers after installing an appropriate application.

As might be expected, the concept of the code has therefore been revisited to exploit fully the hardware that has matured in recent years. The first color two-dimensional codes have emerged. Their purpose is not aesthetic; instead, they seek greater information density and therefore greater data-storage capacity over a given surface. Microsoft HCCB [1] belongs to this new type. Its components are rows of colored triangles using palettes of two, four, or eight colors and thus representing one, two, or three bits of information per module. In this respect Microsoft technology introduces innovations over older codes, but it has the disadvantage of being proprietary.

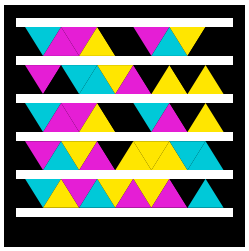

Figure 1.4: An example of a Microsoft HCCB code: Microsoft Tag.
Editorial note (2026): This author-created schematic was independently drawn for the digital edition; its internal color sequence differs from the historical sample. Technical reference: Microsoft HCCB.

This work presents the design of a high-capacity color code based on Quick Response and attempts to extend the standard that defines it. It is important to pursue code innovation along a path different from Microsoft's, both to avoid proprietary technologies and because the structures of older two-dimensional codes such as Quick Response appear excellent and in no way inferior to that of Microsoft HCCB itself.

The idea is therefore to revise the QR Code standard. Its substantial reliability, capacity, and decoding-speed characteristics made it the dominant code in Japan, where it was developed, and secured its broad adoption elsewhere in the world.

The remainder of this section says more about codes mentioned only briefly so far, describing at least one from each class before treating Quick Response in detail.

1.2.3 Stacked 2D Barcodes: PDF417 and Codablock

Stacked two-dimensional barcodes are essentially the most direct evolution of conventional barcodes. They may be seen as a series of very small one-dimensional barcodes

stacked one above another (stacked symbologies). One example is PDF417, whose name indicates that the elementary data unit of the code, called a "codeword," consists of four bars and four spaces occupying a total of 17 modules. PDF417 has good capacity and can encode approximately 2,000 characters in one symbol.

Figure 1.5: Structure of a PDF417 code.
Editorial note (2026): This symbol was independently generated and annotated by the author for the digital edition; it does not reproduce a figure from the standard. Technical reference: ISO/IEC 15438.

The symbol contains from three to 90 rows, each resembling a small barcode. Every row begins with a control codeword that specifies the row number and the error-correction rate being used for it. PDF417 is widely used in transportation and, for example, in mail handled by the United States Postal Service.

Codablock is another stacked two-dimensional barcode. Developed in Germany in several versions, Codablock consists of multiple rows of conventional codes such as Code 39 or, in its more recent Codablock F version, Code 128. Each row may contain up to 64 data characters plus five control characters. Although it is used to a reasonable extent in Germany, it has not achieved broad adoption in the rest of the world.

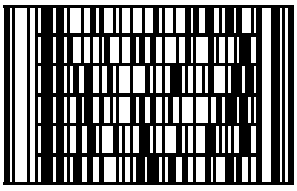

Figure 1.6: Example of a Codablock F code.
Editorial note (2026): This symbol was independently generated by the author for the digital edition and does not reproduce the historical raster sample. Technical reference: AIM Codablock F.

1.2.4 Two-Dimensional Matrix Codes

Two-dimensional matrix codes have a structure quite different from barcodes, although they are sometimes incorrectly described as such. Data are not encoded through bar widths and distances but through the luminance of code areas, typically square, that form the so-called cells (or modules). Unless they perform control functions, these usually carry one bit of information. Modules are therefore the elementary

units on which the structure of the symbol is based. Although these technologies remain in use, they are certainly not recent, having originated in the 1990s.

Data Matrix

One widely used matrix code is Data Matrix, a two-dimensional code composed of black and white modules arranged within a rectangular or square pattern.

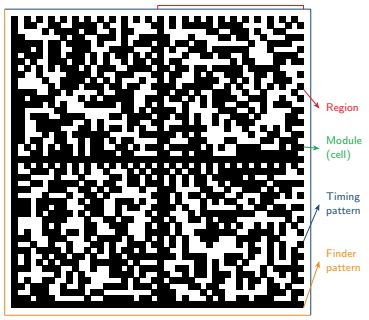

Figure 1.7: Structure of a Data Matrix code.
Editorial note (2026): This symbol was independently generated and annotated by the author for the digital edition; it does not reproduce a figure from the standard. Technical reference: ISO/IEC 16022.

As Figure 1.7 shows, the symbol modules lie within a perimeter formed by finder and timing patterns. Two adjacent uniformly colored borders form the finder pattern; the other two adjacent borders, also forming an L, alternate black and white cells to form the timing pattern.

The finder pattern enables optical readers to locate and orient a Data Matrix code correctly within a scanned image. Even if the scan contains disturbing elements, as it inevitably may because of human error, the optical reader must discard extraneous elements and identify only the scan area constituting the code, whether the code is upside down or has another orientation.

The timing pattern reveals the number of rows and columns in the symbol and therefore provides a key for reading its interior. Knowing only the size of the code area as perceived by the optical sensor would provide no information about the size of one module. The alternating black and white modules of the timing pattern overcome this obstacle and allow decoding to proceed. If the number of modules in the code were fixed, a timing pattern would not be required to determine how many modules make up the symbol, but such a restriction would be limiting. As the amount of data to be encoded increases, so does the number of rows and columns: code dimensions range from 8 by 8 to 144 by 144 cells.

Within the borders formed by these two patterns lie the row-and-column modules that represent encoded data. The cells forming the finder and timing patterns, by contrast, serve control functions and carry no information bits.

Data Matrix symbols offer very good capacity, encoding data ranging from a few bytes to about two kilobytes. It is reasonable to ask how adequate reliability can be

guaranteed for such substantial quantities of data. All symbols using the ECC200 error-correction system employ a sophisticated Reed–Solomon-based mechanism that can reconstruct a proportion of erroneous data reaching as high as 30 percent, which is impossible for conventional linear barcodes. This capability naturally has a cost: the more powerful the desired error correction, the more redundant data must be encoded in the symbol, consuming surface that could otherwise encode additional data.

Quick Response (QR)

The preceding discussion of Data Matrix makes clear that it has an excellent, high-quality structure. This work nevertheless chose to adopt and extend another type of code, Quick Response. Selecting a code to rework and redesign into a high-capacity code exploiting chromatic information necessarily required consideration of code quality. The best result would be obtained by working with the code that, more than others, offered high capacity, reliability, and decoding speed, in addition to broad adoption and popularity.

Although Data Matrix was an excellent candidate, Quick Response also has finder and timing patterns—different in shape from those of Data Matrix but performing analogous functions—as well as a sophisticated error-correction system, high reliability, rapid reading, and broad worldwide popularity. The remainder of this thesis treats Quick Response codes extensively. They are mentioned here only to recall both that QR is a two-dimensional matrix code and that, although it was selected from among all existing codes as the reference for this work, the Data Matrix code just discussed would also have been a good candidate.

1.2.5 Color 2D Codes: Microsoft HCCB

This discussion has noted several times that the technologies considered are far from recent. Yet seeking innovation in an area where research had lain dormant for a long period is not so unusual when Microsoft itself has in recent years promoted a proprietary code called High Capacity Color Barcode (HCCB) [1, 2]. As its name suggests, HCCB uses chromatic information to achieve high data-storage capacity.

Codes of this kind should not be regarded as present-day competitors to established traditional codes, which are highly reliable in applications such as transportation and warehouse management. They do, however, provide a good way to represent large quantities of data optically in a small area while remaining reliable because of advances in modern optical-sensor hardware. It should be recalled that today every smartphone is potentially a good-quality optical reader, given the excellent cameras available on such devices.

HCCB uses groups of colored triangles rather than the square cells discussed for conventional two-dimensional codes such as Data Matrix. Although black and white codes can be generated, information density can be increased by using four- or eight-color schemes. These respectively double or triple the amount of data that can be

encoded in a given area compared with the basic black and white case—or, more generally, any two-value scheme.

Microsoft states that it has encoded 3,500 alphanumeric characters per square inch using fairly ordinary printers and optical sensors.

Irrespective of its actual performance, assuming it is as good as claimed, Microsoft has recently promoted a four-color implementation of HCCB arranged as a 5 by 10 grid of triangles and called Microsoft Tag, which is therefore a particular form of HCCB. Because it can carry only 13 bytes of data, including the redundancy required for Reed–Solomon error correction, this code is essentially a machine-readable web link. When the optical reader decodes the code, the application sends the information it has read over the network to Microsoft's servers, which return the associated URL; the user's browser can then navigate to the relevant website.

This mechanism can consequently be interpreted as placing in a Microsoft Tag only the identifier of content that must in practice be retrieved over the Internet. What, then, was gained by using for such a scheme a code that originated in a project intended to increase information density through color? In practical terms, unless commercial motivations are considered, a system of this kind could have been implemented with codes that had already existed for many years. Although the resulting printed code is smaller than a black and white code, encoding only 13 bytes minimizes that advantage.

Figure 1.4 in fact showed an HCCB instance that was a Microsoft Tag. Its strongly horizontal structure consists of five rows of ten triangles, separated by dedicated white lines. These perform control functions, such as detecting code orientation, and carry no information.

Four- and eight-color code versions other than Microsoft Tag can naturally encode more information, as shown in Figure 1.8.

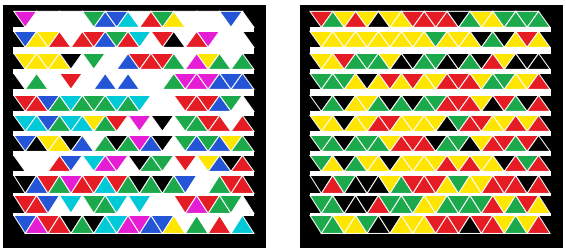

Figure 1.8: Four- and eight-color Microsoft HCCB codes.

Editorial note (2026): The geometry and palette follow the subject discussed in the 2010 thesis, but the author changed twelve individual triangle colors in each panel for this edition. Technical reference: Microsoft HCCB.

Chapter 2

The Quick Response Code (ISO/IEC 18004)

Editorial note (2026)

As a prudential editorial choice, this edition omits Chapter 2 of the 2010 thesis, which describes QR Code specification elements in detail.

The color format discussed in the 2010 thesis—called *High Capacity Colored Two-Dimensional Codes* (HCC2D) only in publications subsequent to the thesis—reuses the square-matrix structure of QR Code: finder patterns, alignment patterns, timing patterns, format information, version information, mask formulas, and Reed–Solomon error-correction behavior compatible with QR Code / ISO/IEC 18004:2006, except for the color-specific rules described in the following chapters.

The following chapters identify those structures only where needed and describe the work specific to the color format, called HCC2D in subsequent publications.

Chapter 3

High-Capacity Color Quick Response Codes

Editorial note (2026): The 2010 thesis used several descriptive expressions for the proposed symbology, including "High-Capacity Quick Response codes," "High-Capacity Color Quick Response Codes," "color QR/Quick Response codes," their high-capacity, higher-capacity, and high-density variants, and corresponding references to the new color symbology, symbols, reader, generator, and software modules. The designation "HCC2D" was adopted in publications appearing after the thesis defense. Throughout this edition, all these historical expressions refer to the format later named HCC2D. Their references to QR/Quick Response are purely descriptive: "QR Code" is a registered trademark of DENSO WAVE INCORPORATED in Japan and other countries, and their use does not imply sponsorship, endorsement, or affiliation, or that the proposed symbology is itself a QR Code.

This is a central chapter of the thesis. Its purpose is to conceive and design an evolution of the Quick Response ("QR") code standard that increases the information density of the encoding area by using chromatic information, rather than evaluating luminance alone as standard QR Codes do. The chapter therefore proposes a high-capacity color two-dimensional code [4] derived from the structure of Quick Response codes.

As the preceding chapters have shown, standard Quick Response codes provide excellent reading speed, reliability, and capacity. Renewed interest in barcode technologies stems in part from the fact that dedicated reading hardware is no longer required: excellent and inexpensive cameras are now available on most mobile devices. This trend shows no sign of slowing, as better cameras are produced at low cost every year.

In this context, the work seeks to use the capabilities of such hardware more fully by redesigning codes that remain relevant but were not conceived for present-day scenarios. When the codes were designed and implemented, it would have been impossible to predict the proliferation of devices such as today's smartphones: in the 1990s, even the first expensive mobile telephones were only just entering the market.

In recent years Microsoft introduced the High Capacity Color Barcode (HCCB), discussed in Section 1.2.5. It represents an initial attempt to use chromatic information to increase the information density of a code. It is, however, a proprietary solution

and does not appear to possess strengths capable of overshadowing well-structured two-dimensional codes such as Data Matrix or Quick Response. This chapter explains how the Quick Response code can be reworked and adapted to exploit chromatic information for greater symbol capacity while retaining sufficient reliability from the excellent structure of standard QR Codes.

3.1 Project Objectives

Defining a new two-dimensional symbology based on Quick Response codes, and stating that color will be used to achieve high information density, does not fully describe the project's objectives. Although that goal remains primary, the project seeks a structure as close as possible to standard QR Codes, so that the new symbols may be regarded as a direct evolution rather than as competitors.

More specifically, because the new codes extend and directly evolve standard QR Codes, the set of standard QR Codes should be a subset of the new symbology. Put another way, a standard QR Code should be a special case of a high-density color QR Code that uses only two colors. Defining the new set in this way provides an important benefit: backward compatibility, allowing a high-capacity color QR reader to continue decoding traditional codes.

Finally, and no less importantly, increasing data density must not excessively weaken the reliability or decoding speed characteristic of QR Codes, nor may the new symbology require uncommon, dedicated, or prohibitively expensive hardware. Failure to meet these objectives would make the new symbology unusable in practice.

3.1.1 Project Scope

The tangible results expected from this work are a specification for the structure of the new code symbology and the implementation of its corresponding generator and optical reader. These results must apply to both desktop and mobile environments. As explained throughout this thesis, mobile devices have played a central role in the recent evolution of two-dimensional codes, and it is therefore essential that the new codes be usable in that context.

3.2 Open Problems

Numerous problems arise when attempting to evolve the Quick Response symbology. Every proposed element or modification must be considered carefully, because new functionality could come at the expense of existing, effective features. In the worst case it could invalidate processes critical to symbol decoding and render the entire evolution pointless: high-capacity but nearly unreadable symbols would clearly be a step backward.

The first question is what must be reworked to introduce color into QR Codes and what can remain exactly as defined by the standard. Modifying an existing element may also make a related concept ill-defined. Even a minimal change can therefore create important theoretical challenges, as often occurs when a complex product such as the Quick Response symbology is altered. This section examines the components of the QR structure and considers possible directions for the new specification, with the specific aim of exposing complications that might arise. The selected solutions are explained in subsequent sections.

3.2.1 Observations on Function Patterns

We begin with the function patterns. The preceding chapter discussed the position, timing, and alignment patterns that form the structural elements of standard QR Codes. How does the introduction of color affect these elements? Do they remain valid, are their functions impaired, are they sufficient as a whole, or must something new be introduced?

Using chromatic information to increase information density is essentially orthogonal to the image-processing algorithms that locate and normalize a scanned symbol, often even in the presence of substantial distortion. The support that standard function patterns give those algorithms should therefore remain effective after colors are introduced. This suggests preserving the patterns for at least two reasons: doing so contributes to backward compatibility, and their structure is the product of extensive experience and provides excellent reliability. These characteristics are precisely why the new color symbology is based on Quick Response rather than on another code.

Although existing patterns should be inherited, the new requirements may call for additional control elements. In particular, a pattern that helps counter chromatic distortion in the image perceived by the optical reader may be useful. Codes with four or more colors are considerably more sensitive to that phenomenon than black-and-white symbols.

3.2.2 Observations on the Encoding Area

The encoding area is divided into the data area, format information, and version information; the preceding chapter examined each region in detail. Because the main effects of introducing color for higher capacity will be concentrated here, what conclusions can be drawn about its evolution?

Modifying the structure of the format and version information, where present, would probably make little sense. First, it would move away from backward compatibility and make it difficult to regard standard QR Codes as a strict subset of the new symbology. Nor is this the only concern. Attempting to reduce the area allocated to this information by exploiting chromatic density would offer little benefit when reliability is far more important. These regions encode few bits, but reading them incorrectly can invalidate the entire decoding process. A minor gain is not worth

such a substantial risk, so their specifications should be inherited directly from the standard and represented only by contrasting black and white elements.

The third field, the area allocated to "Data and Error Correction Codewords,"[1] is where the data of interest are placed. Because error rates are inevitably high when measurements are taken from a physical layer by a hardware sensor—in this case the optical reader—the desired data cannot be encoded alone. Redundancy for error handling must be added in the form of error-correction codewords.

The complete surface assigned to this redundant data will be visibly different from that of standard codes. It will contain elements with contrasting chromatic hues instead of black and white cells. Each module will represent two, three, or four bits rather than one, using palettes of four, eight, or sixteen colors respectively.

3.2.3 Principal Problems and Challenges

Moving away from the standard raises questions about how the ideas should be refined into a consistent specification. This section focuses on the principal challenges rather than presenting an exhaustive list. It identifies the questions that properly arise when modifying a product as complex as the Quick Response code; the sections that follow propose possible answers.

1. Tables for standard QR Codes relate quantities such as total codewords, symbol version, error-correction level, and the Reed–Solomon blocks into which the data area is divided. They naturally omit the number of bits represented by each module, because each standard cell always represents one bit. The proposed symbology is sensitive to this new quantity: symbol capacity rises with the cardinality of the color palette. These tables must therefore be extended to account for the new parameter.

2. To preserve backward compatibility, the structure of existing regions and patterns should not be changed. This leaves no place to state how many colors a symbol uses—two, four, eight, or more—although the reader must know that value to interpret the modules correctly. Should a dedicated field be introduced?

3. Even if the reader learns the palette cardinality, it still does not know which colors the generator used. The encoding palette must serve as a decoding reference. Must those colors be known by the reader in advance, and how can chromatic distortion caused by the camera be handled?

4. Data masking, which improves code reliability, is no longer fully defined once colors are introduced. Certain unfortunate inputs can produce poorly readable symbols. One example is a code with an unreasonable proportion of black to white cells: a symbol that is dark over 90% of its surface is much less reliable

[1] Within the Quick Response standard, a codeword is effectively one byte, or eight bits.

than one whose light and dark areas are balanced at roughly 50% each. The preceding chapter discusses masking more fully; in simplified terms, the standard establishes the following procedure:

- Generate eight matrix mask patterns with the same number of cells as the symbol. A module is dark when the relevant condition is true—for example, $(row + column) \bmod 2 = 0$ or $column \bmod 3 = 0$—and light otherwise. Each of the eight alternatives is associated with its own generating condition.
- Evaluate the eight results obtained by XORing each mask with the unmasked symbol, and select the best one. The standard defines explicit criteria that assign a penalty score to each candidate.

During decoding, measured luminance values are first assigned to the domain {dark, light}, then mapped to {0, 1} through analog-to-digital conversion. The luminance thresholds that distinguish light from dark cells are adaptive. The resulting values can then be XORed with the generated mask pattern to recover whether each module was originally light or dark.

If a module in the proposed high-capacity code represents more than one bit, how should masking operate? In a multicolor palette, what should replace the standard inversion between black and white?

5. How should distances among hues be calculated when the reader must recognize them? Will recognition remain precise enough in a low-quality image to decode the symbol when colors are similar? A color-similarity metric is needed to make the concept of similarity concrete.

3.3 Proposed Design Specifications

Several solutions were found for the problems raised by a possible evolution of QR Codes into high-capacity color symbols. They are certainly not the only possible solutions, but they seek to meet the objectives in Section 3.1. As a whole, the new specification should:

- follow the standard as directly as possible, inheriting every feature that does not strictly need to be reworked for the new requirements;
- favor simplicity, so that evolving a complex and already effective standard does not introduce inconsistencies or compromise its performance.

3.3.1 Extending the Quick Response Tables

In response to problem 1 in Section 3.2.3, the number of bits encoded by each module must be considered. The most direct approach is to extend the tables in ISO/IEC 18004, which relate total codewords, symbol version, error-correction level, Reed–Solomon block structure, and other quantities, but omit bits per module because a standard QR module always represents one bit.

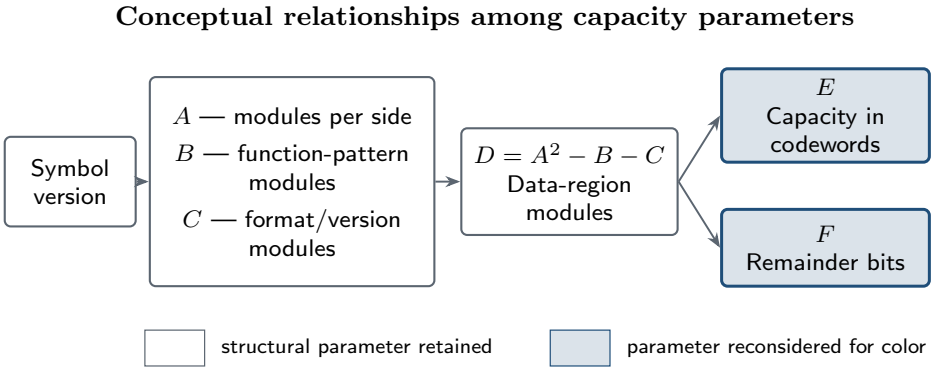

Figure 3.1: Conceptual relationships among the capacity parameters discussed in the 2010 text.

Editorial note (2026): The numerical table in the 2010 source figure is not reproduced in this public digital edition. To make the later references intelligible, its columns, from left to right, were: (1) Version; (2) No. of Modules/side (A); (3) Function pattern modules (B); (4) Format and Version Information modules (C); (5) Data modules except (C), $D = A^2 - B - C$; (6) Data capacity [codewords] (E); and (7) Remainder Bits. In the new diagram, retained Columns 1–5 are shown in white; Columns 6–7, proposed for parameterization, are shown in light blue. The new diagram shows only the conceptual relationships, without table values. Technical reference: ISO/IEC 18004:2000.

Figure 3.1 shows an excerpt from Table 1 of the reference document for the Quick Response standard. For each version, it gives:

1. the number of modules along one side of the symbol matrix; Version 1, for example, has 21 modules per side and therefore $21 \times 21 = 441$ modules in all;
2. the number of symbol modules used by function patterns and thus carrying no information bits;
3. the number of modules representing version and format information;
4. the number of modules in the encoding region that represent the data of interest, including the redundancy required for error correction;
5. the number of representable codewords, each equivalent to one byte; because redundancy is included, the maximum user-data quantity may be much smaller;
6. the number of unusable remainder bits, which occur when the information capacity in bits is not a multiple of eight.

The fields in the table just discussed should therefore be updated to account for the additional parameter given by the number of bits represented by each module. It is necessary to decide which fields should be extended and which should be retained; Figure 3.1 highlights the columns for which parameterization is proposed.

The first attributes in the table remain valid: the number of modules in the symbol, the values for function patterns, and the areas for version and format information can all be retained. As already stated, the intent is to inherit everything not affected by the introduction of the new symbology's color palettes.

Capacity, shown in column E, should instead increase with the palette size. This is the basis of the proposed evolution, and it can be expressed by multiplying by the number of bits represented by each module.

The remainder-bit column can likewise be multiplied by bits per module. If that product reaches or exceeds eight, the bytes thereby obtained could instead be added to the previously calculated data capacity:

$$RemainderBits_{new} = RemainderBits_{old} \times BitsPerModule$$
$$DataCapacity_{new} = DataCapacity_{old} \times BitsPerModule$$

or, minimizing waste:

$$RemainderBits_{new} = (RemainderBits_{old} \times BitsPerModule) \bmod 8$$
$$DataCapacity_{new} = (DataCapacity_{old} \times BitsPerModule) + \frac{RemainderBits_{old} \times BitsPerModule}{8}.$$

To simplify table parameterization, and because the second solution provides only a minimal benefit, the discussion that follows adopts the first assumption.

Symbol Character Count and Input-Data Capacity

Figure 3.2 presents a table relating each code version to the corresponding data capacities for the numeric, alphanumeric, 8-bit byte, and Kanji modes. The preceding chapter describes these Quick Response modes and the related information in greater detail.

Conceptual dependencies of input capacity

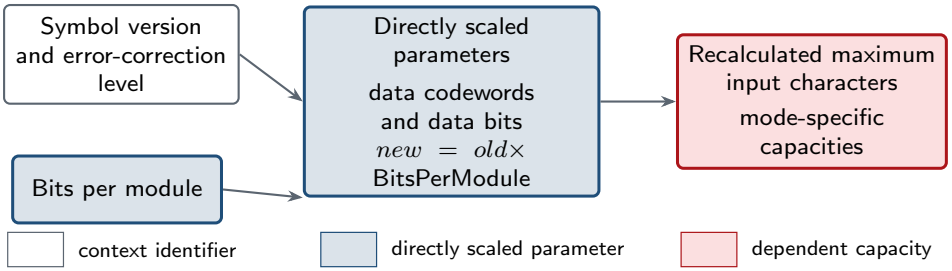

Figure 3.2: Conceptual dependencies among version, error correction, data size, and input capacity.

Editorial note (2026): The numerical table in the 2010 source figure is not reproduced in this public digital edition. Its columns, from left to right, were: (1) Version; (2) Error correction level; (3) Number of data codewords; (4) Number of data bits; then, under Data capacity, (5) Numeric; (6) Alphanumeric; (7) 8-bit Byte; and (8) Kanji. Columns 1–2 were white and served as identifiers. Columns 3–4, shown in light blue in the new diagram, were subject to direct modification. Columns 5–8, shown in pink, were recalculated as a consequence of the former. The new diagram retains only this conceptual dependency, without table values. Technical reference: ISO/IEC 18004:2000.

This table must also be parameterized to support higher-capacity color Quick Response codes. In general, the tables are being updated rather than rebuilt from scratch so as to pursue a direct evolution that remains backward compatible with standard codes. In Figure 3.2, columns with a colored rather than white background identify areas that require modification. Unlike Figure 3.1, where some columns were inherited directly, every field here must be updated in some way. Two different colors also emphasize that the columns in the right-hand region, with a pink background, must be extended mainly because they depend on the columns in the left-hand region, with a blue background, which are modified directly. The data capacities for the various modes must consequently be updated as well.

Bits per module can again be incorporated with a simple multiplier:

$$NumberOfDataCodewords_{new} = NumberOfDataCodewords_{old} \times BitsPerModule$$

$$NumberOfDataBits_{new} = NumberOfDataBits_{old} \times BitsPerModule.$$

The data-capacity columns for numeric, alphanumeric, 8-bit byte, and Kanji modes must then be recalculated from the new number of data codewords.

Example. In alphanumeric mode, two characters are stored in eleven bits, so the corresponding data capacity is approximately $((NumberOfDataCodewords \times 8)/11) \times 2$. Once the data-codeword and data-bit fields change, capacity is updated using the same calculations that the standard ordinarily applies to each mode.

Error-Correction Characteristics

We now consider another table from ISO/IEC 18004, this time addressing error correction, a function fundamental to the reliable use of two-dimensional codes.

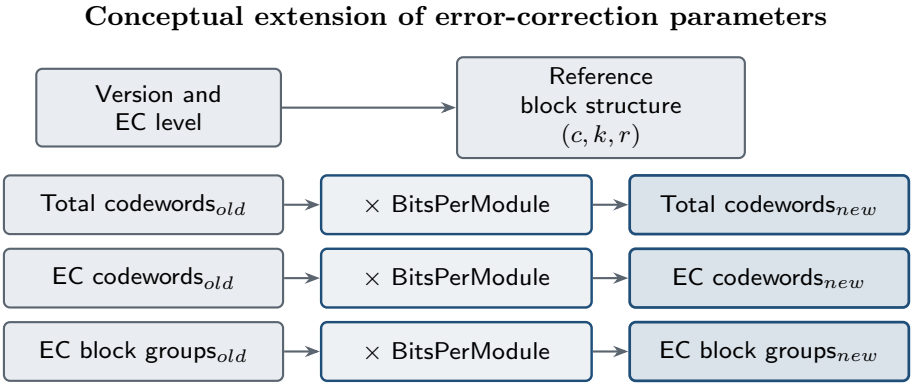

Figure 3.3: Conceptual relationships used to extend the error-correction parameters.
Editorial note (2026): The numerical table in the 2010 source figure is not reproduced in this public digital edition. Its columns, from left to right, were: (1) Version; (2) Total number of codewords; (3) Error correction level; (4) Number of error correction codewords; (5) Number of error correction blocks; and (6) Error correction code per block. Columns 2, 4, and 5 had a gray background and were to be parameterized; Columns 1, 3, and 6 had a white background and were considered directly inheritable. The new diagram shows only the conceptual flow; the example values remain in the text. Technical reference: ISO/IEC 18004:2000.

As before, the proposed parameterization reduces to a simple multiplier; the quantities whose values were considered unsuitable for the proposed evolution of Quick Response codes are therefore redefined as follows:

$$TotalCodewords_{new} = TotalCodewords_{old} \times BitsPerModule$$

$$ErrorCorrectionCodewords_{new} = ErrorCorrectionCodewords_{old} \times BitsPerModule$$

$$ErrorCorrectionBlocks_{new} = ErrorCorrectionBlocks_{old} \times BitsPerModule.$$

So far the discussion has stated how the tables can be updated without giving a concrete demonstration. The following example is not intended as a formal proof, but provides reasonable confidence that the reworking is consistent.

Example. For Version 5 at level H, shown in Figure 3.3, there are 134 total codewords: 88 ECC codewords providing error-correction redundancy and 46 data codewords. Each of the four Reed–Solomon blocks has 22 ECC codewords. The first two blocks contain 11 data codewords each, while the other two contain 12.

For clarity, "data codewords" here means the nonredundant information of interest, whereas ECC codewords are the redundancy needed for error correction and reliability. This clear distinction helps explain the encoding logic, but the final representation within the symbol does not separate or distinguish the two kinds. A single region contains all redundantly encoded data; separate subregions would undermine reading reliability.

The relationships among the tabulated values are therefore:

$$4 \times 22 = 88 \textit{ ECC codewords}$$
$$11 \times 2 + 12 \times 2 = 46 \textit{ data codewords}$$
$$88 + 46 = 134 \textit{ total codewords}$$

The error-correction capacity of each block is given by the final value in the triples in the last column of Figure 3.3; the four blocks can therefore correct at most $11 \times 4 = 44$ codewords. The ratio between this maximum and all codewords is $44/134 = 32.8\%$. Level H requires the symbol to correct approximately 30% of its total codewords; it is the highest level and uses the most redundancy.

Consider now an extended Version 5 level-H symbol using four colors, or two bits per module. Under the proposed parameterization it has 268 total codewords, divided into 176 ECC codewords and 92 data codewords. Each of eight Reed–Solomon blocks contains 22 ECC codewords; the first four blocks contain 11 data codewords and the other four contain 12:

$$8 \times 22 = 176 \textit{ ECC codewords}$$
$$11 \times 4 + 12 \times 4 = 92 \textit{ data codewords}$$
$$176 + 92 = 268 \textit{ total codewords.}$$

The eight blocks can correct $11 \times 8 = 88$ codewords. The ratio is again $88/268 = 32.8\%$. The simple multipliers therefore preserve error-correction levels L, M, Q, and H in the proposed higher-capacity evolution, an important condition for the objectives in Section 3.1.

The Character Count Indicator

The parameterized tables appear consistent, but a QR symbol is complex, and extending those tables indirectly creates an inconsistency in mode handling: the Character Count Indicator must also be addressed. It states how many characters, in accordance with the selected mode, are encoded in the symbol. The proposed extension may allow more characters than a given symbol version and mode ordinarily support. If too few bits are assigned to the indicator, the required count may no longer fit.

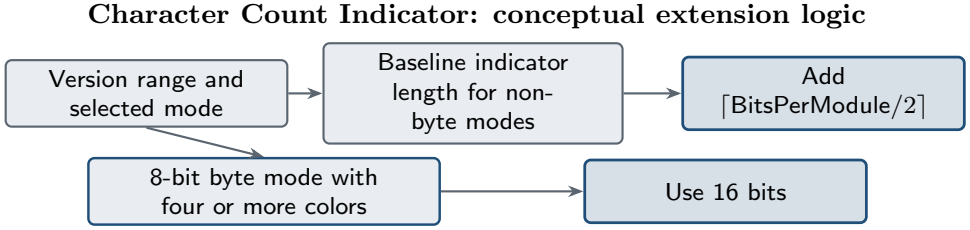

Figure 3.4: Conceptual extension logic for the Character Count Indicator.

Editorial note (2026): The numerical table in the 2010 source figure is not reproduced in this public digital edition. Its columns, all with a white background, were, from left to right: (1) Version; (2) Numeric Mode; (3) Alphanumeric Mode; (4) 8-bit Byte Mode; and (5) Kanji Mode. Its rows grouped Versions 1–9, 10–26, and 27–40. The new diagram shows only the extension logic; subsequent 2010 references to bits readable in the figure concern the omitted values. Technical reference: ISO/IEC 18004:2000.

Figure 3.4 shows the bits allocated to the Character Count Indicator. The standard optimizes the original values to the point of saving even a single bit. Although beneficial in itself, that optimization causes a problem for the proposed evolution: four-, eight-, and sixteen-color palettes can represent approximately twice, three times, and four times as much information as a standard code, and the indicator consequently needs at least one or two additional bits.

Example. Versions 1 through 9 allocate nine bits to the Character Count Indicator in alphanumeric mode, representing values up to 511. With the greater density provided by color, such symbols may contain more than 511 alphanumeric characters, which the existing indicator cannot express.

The table can be extended according to the following criteria:

- For 8-bit byte mode, assign 16 bits regardless of the version whenever a palette of four or more colors is used.[2]
- For all other modes, parameterize the table as follows:[3]

$$CharacterCountIndicatorLength_{new} = CharacterCountIndicatorLength_{old} + BitsPerModule/2.$$

3.3.2 Managing the Introduction of Color into Quick Response Codes

Problem 2 in Section 3.2.3 requires backward compatibility while leaving the reader without information about how many colors the new symbol uses—two, four, eight, or sixteen. The reader must obtain that information for decoding to proceed.

Problem 3 requires the reader to learn which colors the generator actually used. This palette is central to interpreting the colored modules, so either generator and reader must agree on it in advance or another mechanism must be found. Even then, camera-induced chromatic distortion is unavoidable and must be managed if the higher-capacity color QR Code is to work in real conditions.

In summary, while preserving compatibility with standard QR Codes, the symbol must express how many colors—and therefore how many bits per module—were used. Must the decoder know those colors in advance, and how can a camera image be prevented from significantly changing the expected hues?

Proposed Solutions

The proposal introduces a new field, or pattern, that does not conflict with existing QR elements: a color palette used as a legend.

[2]For black-and-white Quick Response codes in 8-bit byte mode, continue to use eight bits for versions up to and including Version 9.

[3]For odd values of $BitsPerModule$ other than one (thus excluding standard QR Codes), round $BitsPerModule/2$ upward to the next integer.

This resolves many chromatic-distortion problems at their source. If the colors of the data-plus-ECC modules are altered, the legend modules are altered in the same way, preserving decoding within reasonable limits. If a capture makes colors nearly identical, the problem remains and decoding fails.

The number of colors must still be conveyed, and the decoder should decide when a poor scan warrants immediate failure and a new capture. Under a fail-fast approach, there is little value in continuing computation when the observed colors are too distorted to recognize. Some chromatic alteration can be tolerated, but only within defined limits.

A simple and efficient rule is to place dark colors first and light colors after them in the legend, or vice versa. With an expected luminance order, the decoder can check whether each color is at least broadly as light or dark as expected. If so it can continue; otherwise it should fail.

Replicating the legend avoids making it a fragile bottleneck and improves chromatic precision. A legend that violates the luminance criteria during decoding should be discarded: if a dark colored module is read as light, or vice versa, its sampled color is unacceptable. Among the remaining valid legends, equivalent module samples can be averaged.

This solution frees the decoder from knowing the colors in advance. The encoder can choose them, subject only to an increasing or decreasing luminance order. The decoder can also determine the number of colors without a separate field by examining the replicated legends and their ordered modules.[4]

Replicated legends can occupy several positions, but they must not compromise backward compatibility. They could be placed outside the square symbol or inside it without damaging existing patterns. Extension Patterns would be the cleanest location because they were designed for future QR functions, but they are permitted only in the deprecated and discouraged Model 1 symbology.

[4]For reading reliability, a balance between light and dark regions, and the masking process, the proposal further requires exactly half the legend colors to be light and half dark. This concept is clarified below.

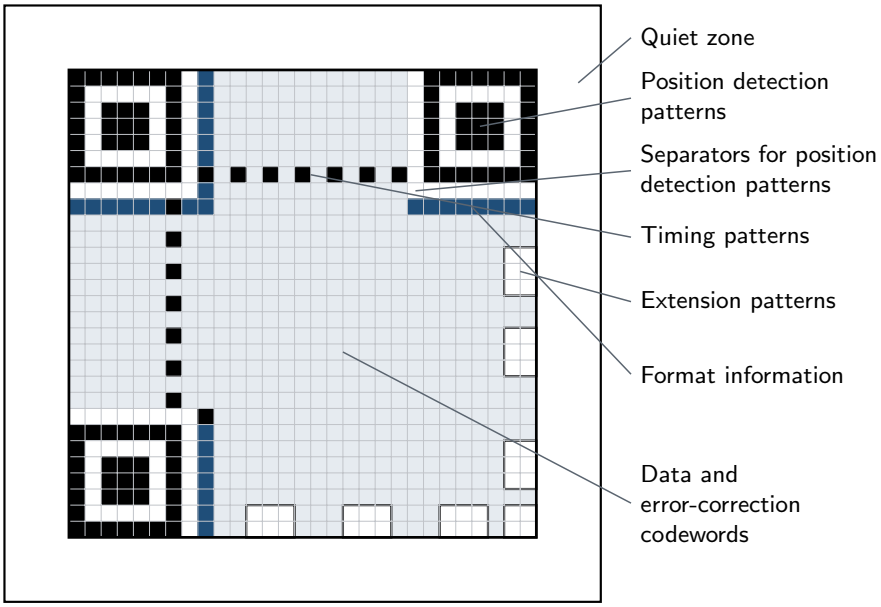

Figure 3.5: Author-created schematic of the functional regions of a Model 1 symbol.
Editorial note (2026): This schematic was created by the author for the digital edition and does not reproduce a figure from the standard; technical reference for the discussion: ISO/IEC 18004:2000.

As an initial proposal, the legends are placed next to and outside the square representing the QR Code. A more refined placement can be developed in future, but this solution is sufficient for an adequately reliable first design.

3.3.3 Reworking the Concept of Data Masking

The eight masks in the Quick Response standard principally increase code reliability. More specifically, they assist optical reading by preventing an unreadable symbol and by producing a higher-contrast representation of the data.

For example, masking balances light and dark areas and prevents their proportions from straying too far from 50%–50%. It also prevents accidental patterns resembling position detection patterns from appearing in the data-plus-ECC area. Such patterns can occur for rare input values and would make an unmasked code unreadable.

Luminance must therefore be considered even when the data-plus-ECC modules have several hues. Without a revised masking concept, luminance patterns in the data area might simulate position detection patterns and make the code effectively unreadable.

Problem 4 in Section 3.2.3 asks how masking should proceed when a module represents several bits, a case not contemplated by the standard. Where standard masking exchanges black and white, a multicolor palette requires a definition of which colors invert to which others. The solution should adapt the existing effective masking process without overturning its structure.

First, treat a color QR Code as a binary matrix by classifying the color at each module position as light or dark. The traditional algorithm can then evaluate the eight masks on that matrix and select the best. In other words, begin with an unmasked color QR Code, map its colored modules to light and dark according to their luminance, and use the resulting ad hoc binary symbol to choose a mask for the original color symbol.

Once the best mask is selected, how is XOR applied when each cell represents multiple bits, and what does it mean to invert a color? For each module, XOR every bit in its sequence with the corresponding single mask value. The sequence either remains unchanged or every bit is inverted.

Example. In a four-color QR Code, consider a module representing the sequence "01." Masking leaves it as "01" or changes it to "10." In the first case, a light module remains light and a dark module remains dark. In the second, the mask requires its luminance to be inverted. The remaining question is which colored module represents that inversion.

Combining the preceding observations leads to the following proposal:

- The first half of every legend contains light colors and the second half dark colors, or vice versa. This 50%–50% division supports a better luminance balance.
- The bit sequence assigned to each color is arranged so that inverting every bit produces the sequence belonging to a color of opposite luminance.

Figure 3.6 illustrates standard XOR-based data masking, including the mapping of matrix bit values to the resulting black-and-white symbol modules.

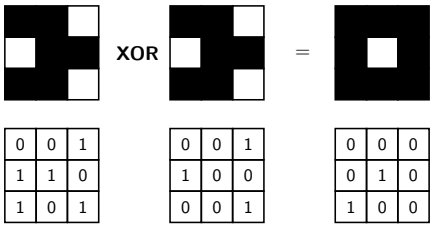

Figure 3.6: Example of data masking under the Quick Response standard.
Editorial note (2026): This example was independently created by the author for the digital edition to illustrate the 2010 explanation; it does not reproduce a figure from the standard.

Figure 3.7 shows how masking was reworked for the introduction of color as an additional information carrier. Direct comparison with Figure 3.6 shows that the mature and effective standard process has been evolved as directly as possible.

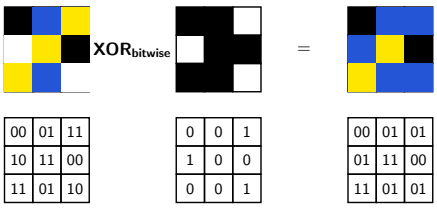

Figure 3.7: Example of data masking reworked for the new symbology.
Editorial note (2026): This example was independently created by the author for the digital edition to illustrate the 2010 proposal; it does not reproduce a figure from the standard.

These examples complete the masking design, but a related and essential issue remains. A colored module is inverted by selecting the color whose bit sequence is obtained by changing every bit of the first sequence; the two opposite sequences must also correspond to opposite luminance classes. The colors must now be organized in the legend, and the opposite pairs selected.

One possibility is to order the entire legend by luminance while retaining separate light and dark halves. This is valid and was tested successfully, but does it emphasize the contrast needed for reliability? The left side of Figure 3.8 shows this total ordering, together with mappings from bit sequences to colors and a table analyzing the contrast between modules. The central colors perform poorly under masking: although they are "opposites" in different luminance groups, their intrinsic luminance difference is very small. In the example, inverting sequence "011" at luminance 0.75 into sequence "100" at luminance 0.65 may therefore be insufficiently effective.

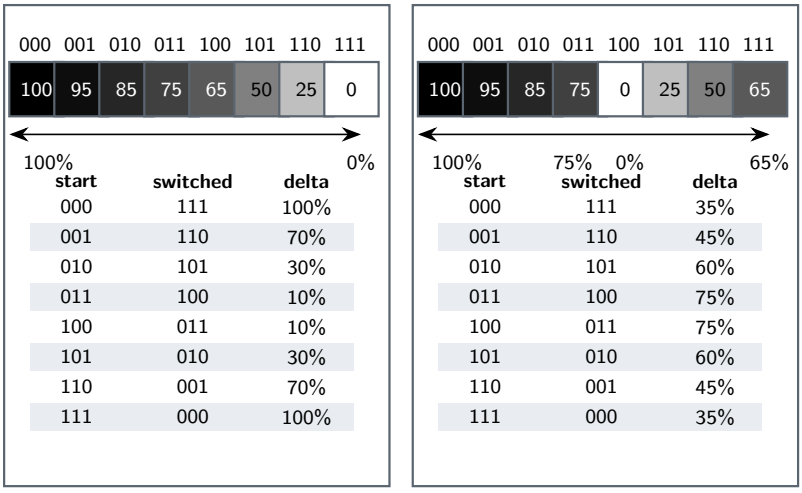

Figure 3.8: Alternative arrangements of colored modules within the legends. Actual colors are replaced by their corresponding luminance values.
Editorial note (2026): This vector comparison was independently recomposed by the author for the digital edition from the values and relationships stated in the 2010 text.

The second alternative, shown on the right of Figure 3.8, distributes luminance

differences more evenly instead of producing high contrast at the extremes and low contrast in the center. It thereby provides sufficient contrast for every pair of opposite modules.

3.3.4 Defining a Chromatic-Distance Metric

Problem 5 in Section 3.2.3 calls for a distance metric among the hues that the reader must recognize. Distinguishing similar colors in an image is critical: inadequate accuracy can prevent the symbol from being decoded and would defeat the proposed evolution of QR Codes.

Fortunately, even the comparatively simple and immature similarity metric used in the first experiments provides enough accuracy for multicolor palettes. An intuitive measure is Euclidean distance in either three-dimensional RGB space (red, green, blue) or YCbCr space (luminance, blue chrominance, red chrominance). Its simplicity limits performance but keeps computational cost low. Although a higher-quality chromatic-distance metric remains future work, this initial measure makes the codes usable and supplies a valid operational definition of color similarity.

Chapter 4

Design and Implementation of the Generator and Reader

This chapter describes the implementation of a generator and a reader for the high-capacity color Quick Response codes [4] discussed in the preceding chapter. Because the symbology must be usable on both desktop and mobile devices, it also introduces Android OS, which was then used on a broad range of recent smartphones and was selected as the mobile-reader platform. The following chapter presents the first experimental results.

The evolution proposed in the preceding chapter was implemented and shown to be viable. Further improvements and optimizations remain useful in both image processing and the reworking of Quick Response components. Nevertheless, the first versions of the encoder and reader can generate, recognize, and decode the new two-dimensional symbology successfully, building on the experience embodied in ISO/IEC 18004.

4.1 Reference Projects

Rapidly implementing generators and readers would have been difficult without the excellent results of existing projects, especially for the demanding image-processing algorithms used in symbol recognition. The color QR generator was based on the open-source libqrencode library[1] [5], which made it possible to produce an application that met the project's needs quite fully. Future generator work could therefore focus chiefly on code structure and fixes.

The reader was made possible by the open-source ZXing library[2] [6], an active

[1]Link: http://megaui.net/fukuchi/works/qrencode/index.en.html
[2]Link: http://code.google.com/p/zxing/

project refining the recognition of several symbologies, including Quick Response. Its most interesting characteristic is perhaps its portability. Although its principal goal was barcode reading on Android, ports were also available for iPhone OS and for some phones supporting Java Micro Edition.[3]

Decoder work presents richer conceptual challenges than encoder work because it concentrates all image processing: detecting codes, managing distortions, sampling module positions correctly, and now also distinguishing their hues.

4.2 Introduction to Android OS

Android OS is an open-source mobile operating system based on the Linux kernel. At the time of this work it had only recently become widespread and appeared to have enormous growth potential. It is managed by the Open Handset Alliance and was effectively promoted by its predominant member, Google. The alliance included hardware producers such as HTC, LG, and Motorola and mobile operators such as T-Mobile and Vodafone, while development of Android itself was handled almost entirely by Google despite the presence of smaller software companies.

Android was clearly intended to compete in a market then dominated by Apple and the iPhone. Taking market share from an established product with broad user approval was a substantial challenge. Android, however, ran on a wide range of smartphones, including less expensive products as well as high-end devices, and could therefore reach users unable to afford the higher cost of an iPhone.

The mature and continually evolving iPhone offered an experience at least as good as contemporary Android. Android 1.5 and 1.6, then the most widespread releases, were not yet mature. Firmware versions 2.1 and 2.2 had been released, but in mid-2010 remained uncommon because updates depended on hardware vendors, which could choose not to provide them. HTC, for example, had announced that 2009 devices such as the HTC Magic and HTC Hero would not receive Froyo 2.2.

Although Android itself was open, unofficial developer teams faced another obstacle: only the manufacturer held mature drivers for hardware such as the camera and accelerometer. Unofficial distributions could therefore struggle to make every hardware component work fully, regardless of the effort invested.

Despite these limitations, Android had compelling qualities for both users and developers. Its openness differed sharply from Apple's proprietary ecosystem; its software could mature rapidly; and the companies promoting it, led by Google, gave it strong prospects for worldwide adoption. Its software development kit was available for Windows, Linux, and Mac OS, whereas iPhone development required Apple hardware. Android was therefore selected not by chance but as the only convincing alternative to iPhone OS at the time, unlike platforms such as Windows Mobile.

[3] At the time, these ports did not reach the quality of the Android implementation.

4.3 Impact of Android OS

Having outlined the development platform, we can consider its effect on the reader. Android is the mobile device's base software and must supply the APIs required by the application.

4.3.1 Code-Portability Limitations

Android development uses Java; performance-critical portions can also be written in C and compiled as native libraries. A Linux kernel might suggest that the GNU C Library and either Java Standard Edition or Java Micro Edition would be available. This is not the case. Android is Linux-based but is not a Linux distribution, and exposes only part of the GNU library set, reducing the portability of code written for Linux. It deliberately omits both Java SE and Java ME, likewise preventing direct portability from other Java environments.

4.3.2 An Emulator Inadequate for the Project

The reader requires an optical sensor. Android applications can access the camera, but the emulator supplied with the SDK did not support it. Many Android applications could be developed entirely with the emulator, but this project could not. The principal difficulty was not merely owning a physical device, but the slower development cycle: every test required reinstalling and launching the application on the device. Diagnosis and debugging were also more difficult than in a desktop environment.

4.3.3 Camera Use: Project Risks and Development Considerations

Modern smartphone cameras were already sufficiently accurate for optical code reading, but practical use and default image formats introduced difficulties. Reading is naturally a repeated-attempt process: the user moves the device over the code until the symbol is recognized. The reader therefore works on a sequence of captures rather than a single image. ZXing[4] uses Camera Preview so that the user sees a live view and can frame the code. Each frame is another opportunity to decode it.

Although these cameras could produce high-resolution images, individual Camera Preview frames were relatively low quality. More importantly, Android's default preview format carried limited chromatic information because it subsampled color. YCbCr 4:2:0 was not necessarily the only available format, but it was the only one every Android device was required to support. To work on every device rather than a few models, the reader therefore had to process semiplanar YCbCr 4:2:0 images.

[4]See Section 4.1, on the open-source reference projects.

YCbCr denotes a family of three-dimensional color spaces used in video and digital photography. Unlike RGB's red, green, and blue components, it uses luminance, blue chrominance, and red chrominance. In YCbCr 4:2:0 each 2×2 pixel block has four luminance samples, one per pixel, but all four pixels share one blue- and one red-chrominance sample. This chromatic subsampling is a significant risk for the reader. It saves space because human vision is much more sensitive to small luminance changes than to chromatic changes, so video compression can discard some color information with limited effect on viewing.

In a planar representation, luminance and the two chrominance components are stored as three separate data matrices which combine into the final image. A semiplanar representation is hybrid: luminance remains separate, while red and blue chrominance share one block with interleaved samples.

Semiplanar YCbCr 4:2:0 uses half the data that would be needed without subsampling. The reader would have benefited from all available chromatic quality, but this platform constraint was outside the project's control.

Android's APIs for Camera Preview frame data were also sparse. No data-handling classes were provided; the developer received only a raw array described by the constant `YCbCr_420_SP`. Despite the name YCbCr, the red- and blue-chrominance data were reversed. Understanding and converting these data to RGB therefore required work without adequate documentation.

As the thesis was being completed, Froyo introduced a relevant improvement: *"New YUV image format API to enable compression from YUV to JPEG and manipulation of YUV data."* Here YUV refers in practice to the YCbCr data of interest, as the terms are often used interchangeably. This support was promising, but most devices still ran Android 1.5/1.6 and many others ran 2.1, while 2.2 had not yet spread. Programming against the minimal API sets of 1.5 and 1.6 (API levels 3 and 4) required greater effort but allowed installation on all Android devices rather than only the newest models. This argument would cease to apply if the update process improved and users moved to current releases.

One workaround was to study internal Android modules written in C. Although applications could not call them, they could guide an equivalent Java implementation. A platform source file[5], for example, enabled a YCbCr 4:2:0 SP to RGB conversion. The reader need not convert the entire image, but it is useful to be able to manipulate the raw capture data when necessary. The following excerpt obtains one pixel in RGB:

```
private RgbPixel getArgbPixel(int x, int y){
        int luminance = (0xff & ((int) yCbCr420SPdata[(topOffset + y) *
            width + leftOffset + x])) - 16;
        if (luminance < 0) luminance = 0;
        int offset = framesize + (width * ((topOffset+y)>>1)) +
            (((leftOffset+x)>>1)<<1);
        /* Android appears to reverse U/V (Cb/Cr): interleaved
         * chrominance data contain red first and blue second. */
```

[5]Source path: platform/hardware/msm7k.git/yuv420sp2rg/yuv420sp2rgb.c

```
        int u = (0xff & yCbCr420SPdata[offset+1]) - 128;
        int v = (0xff & yCbCr420SPdata[offset]) - 128;
        int luminance1192 = luminance * 1192;
        int r = (luminance1192 + 1634 * v);
        int g = (luminance1192 - 833 * v - 400 * u);
        int b = (luminance1192 + 2066 * u);
        if (r < 0) r = 0; else if (r > 262143) r = 262143;
        if (g < 0) g = 0; else if (g > 262143) g = 262143;
        if (b < 0) b = 0; else if (b > 262143) b = 262143;
        int rgb = 0xff000000 | ((r << 6) & 0xff0000) |
                  ((g >> 2) & 0xff00) | ((b >> 10) & 0xff);
        return new RgbPixel(rgb);
}
```

Here `yCbCr420SPdata` is the raw array supplied by the Android camera APIs, `framesize` is the total pixel count, and many numerical values are constants used by the RGB conversion.

4.4 Generator Implementation

The color QR generator reworked the open-source libqrencode project described in Section 4.1, shortening development time. The original C code was produced for Linux, so the modified version was initially compiled in that environment. Cygwin immediately made Windows executables possible by providing a Linux-like environment.

Desktop platform constraints were thus overcome quickly. Mobile support required the Android Native Development Kit, which compiles C sources into native Android libraries using the platform's partial GNU C Library support. That minimal set was sufficient to build an ARM port of the generator rather than the common desktop x86 target. High-capacity color QR symbols could therefore be encoded on both desktop systems and Android devices, while native code retained its performance advantage over interpreted Java instructions.

4.4.1 Source Structure

Libqrencode is relatively small, consisting of eight C source files plus their headers. Its modules are summarized below.

- `bitstream.c`: manages bit streams. Single-bit operations require bit masks; this module defines the `BitStream` structure and its procedures.

```
typedef struct {
        int length;
        unsigned char *data;
} BitStream;
```

A pointer to a character array and an integer length are sufficient.

- `mask.c`: generates the eight QR masks and selects the pattern that produces the most reliable symbol. The following routine evaluates each masked candidate by applying the standard penalty criteria.

```
1  _STATIC int Mask_evaluateSymbol(int width, unsigned char *frame)
2  {
3  int x, y;
4  unsigned char *p;
5  unsigned char b22, w22;
6  int head;
7  int demerit = 0;
8
9  p = frame;
10 for(y=0; y<width; y++) {
11         head = 0;
12         runLength[0] = 1;
13         for(x=0; x<width; x++) {
14                 if(x > 0 && y > 0) {
15                         b22 = p[0] & p[-1] & p[-width] &
                               p[-width-1];
16                         w22 = p[0] | p[-1] | p[-width] |
                               p[-width-1];
17                         if((b22 | (w22 ^ 1))&1) demerit += N2;
18                 }
19                 if(x == 0 && (p[0] & 1)) {
20                         runLength[0] = -1; head = 1;
                               runLength[head] = 1;
21                 } else if(x > 0) {
22                         if((p[0] ^ p[-1]) & 1) { head++;
                               runLength[head] = 1; }
23                         else runLength[head]++;
24                 }
25                 p++;
26         }
27         demerit += Mask_calcN1N3(head+1, runLength);
28 }
29 for(x=0; x<width; x++) {
30         head = 0; runLength[0] = 1; p = frame + x;
31         for(y=0; y<width; y++) {
32                 if(y == 0 && (p[0] & 1)) {
33                         runLength[0] = -1; head = 1;
                               runLength[head] = 1;
34                 } else if(y > 0) {
35                         if((p[0] ^ p[-width]) & 1) { head++;
                               runLength[head] = 1; }
36                         else runLength[head]++;
37                 }
38                 p+=width;
39         }
```

```
40          demerit += Mask_calcN1N3(head+1, runLength);
41  }
42  return demerit;
43  }
```

The first portion detects blocks of same-color modules. An $m \times n$ block adds $(m-1)(n-1)N_2$ to the penalty. Calls to `Mask_calcN1N3` add penalties proportional to N_1 and N_3 for long runs of same-color modules and for configurations resembling finder patterns (the 1:1:3:1:1 light/dark ratio). The latter is particularly serious, so N_3 is much larger than the other constants.

- `qrenc.c`: contains the program's `main` function and command-line argument handling.
- `qrencode.c`: a central encoding module that fills the symbol matrix with codewords from the previously calculated final sequence in the positions prescribed by the QR Code specification.[6]

```
typedef struct {
        int version;          // version of the symbol
        int width;            // width of the symbol
        unsigned char *data; // symbol data
} QRcode;
```

The original structure holds the version, width in modules, and a pointer to memory treated as a matrix. Every element is a function-pattern or data-area module and carries exactly one bit. The color project must place several bits in each module. It retains the standard placement order but reworks the filling procedures and data structures. The modified structure adds the number of bits per module, replaces `data` with one matrix per bit plane, and adds a luminance matrix derived from those planes for color-code masking:

```
typedef struct {
        int bits_per_module;              // number of bits per
            module
        int version;                      // version of the symbol
        int width;                        // width of the symbol
        unsigned char *luminance_data;    // luminance symbol data
        unsigned char **bitmatrices_data;// one matrix per bit
            plane
} QRcode;
```

- `qrinput.c`: handles input, including conversion to a bit stream and determination of the minimum version needed for the data.

[6] For further detail, see Section 2.3.3, on codeword placement in the symbol's data area.

- `qrspec.c`: contains specification data published in the standard's tables, including Reed–Solomon block counts, remainder bits, total modules, and maximum codewords. It was modified according to Section 3.3.1. The original data-codeword lookup is:

```
static int QRspec_getDataLength(int version, QRecLevel level)
{
                return qrspecCapacity[version].words
                       - qrspecCapacity[version].ec[level];
}
```

  The corresponding extended function multiplies the original value by bits per module:

```
int __modded_QRspec_getDataLength(int version, QRecLevel level,
                                  int bits_per_module)
{
                return bits_per_module
                       * QRspec_getDataLength(version, level);
}
```

- `rscode.c`: implements the Reed–Solomon error-correction algorithm.
- `split.c`: divides the sequence according to the encoding modes.

4.5 Reader Implementation

Reworking ZXing made it possible to produce a color QR reader that identified, processed, and decoded the new symbology on both mobile and desktop systems. The open-source project shortened development and likely improved the result.

ZXing is written in Java, which supplies a degree of portability, but the project pursued that goal deliberately because Android's lack of Java SE and Java ME tends to make it a separate environment. The modifications preserve ZXing's portability. They also keep original project code as separate as possible from the added high-capacity QR modules. This permits distinct reader versions to be built from different ZXing releases. Because ZXing remained active and continued to improve, the ideal arrangement was a set of components that could be attached easily whenever a new upstream build appeared.

4.5.1 Project Structure

Portability and separation were achieved through extensive class inheritance and method overriding. Added classes extend original classes and redefine the methods that require adaptation for color-symbol decoding. The reader is much larger than

the generator: several packages contain dozens of source files, and more than twenty ZXing files were modified or added.

The changes redefine existing methods and introduce new image-processing modules for chromatic issues. They were designed to remain valid for standard QR Codes and, where possible, easy to attach to future ZXing versions.

The architecture-neutral core packages are `com.google.zxing`, `com.google.zxing.qrcode`, `com.google.zxing.qrcode.detector`, `com.google.zxing.qrcode.decoder`, and the newly added `com.google.zxing.qrcode.fullcolor`. Platform-dependent packages are `com.google.zxing.client.android` and `com.google.zxing.client.j2se`. An added subclass is identified below as *Mod-*{*original class name*}.

4.5.2 Architecture-Independent Core Packages

Core components implement the application logic independently of the executing device. They must not invoke functions exclusive to Android, Java SE, or Java ME, but only the common subset.

- In `com.google.zxing`, shared by readers for many one- and two-dimensional symbologies:[7]
  - *ModBinaryBitmap* extends *BinaryBitmap* with source-image attributes needed for chromatic processing. The original retained luminance but discarded color, which blocked this work.
  - *ModMultiformatReader* extends *MultiformatReader* to enable the new symbology. It controls which readers are active at run time, such as one-dimensional readers or Quick Response readers.
- In `com.google.zxing.qrcode`, specific to Quick Response:
  - *ModQRCodeReader* extends the core *QRCodeReader*.[8] Its `decode()` method is redefined to invoke the adapted Detector for color symbols, followed by the Decoder reworked according to Chapter 3.
- In `com.google.zxing.qrcode.detector`, the Detector identifies a symbol in an otherwise arbitrary scanner or camera image without assuming its size, orientation, or module luminance:
  - *ModDetector* invokes a redesigned grid sampler and a new module that evaluates the chromatic components of replicated legends when present.

[7] ZXing supports several code families, including EAN-8 and EAN-13 among one-dimensional symbols and QR Code and PDF417 among two-dimensional symbols.

[8] "Main class" refers here only to the core. Platform-dependent portions, such as Android and ordinary desktop systems, have their own main classes, which interact with the core class.

  - *ModDefaultGridSampler* also samples legend positions and converts the sampled chromatic value into a sequence of one to four bits.[9]
  - *ModDetectorResult* is adapted to carry the redesigned Detector's result.
  - *PaletteFinder* is new. It processes sampled legend points to produce reference colors for module quantization. If no legend is present, it returns processing to standard QR decoding.

- The package `com.google.zxing.qrcode.decoder` requires extensive changes to meet the preceding chapter's specification:

  Editorial note (2026): The 2010 Italian text has `com.google.zxing.qrcode.detector` at this point as an evident typographical error; the context and the classes listed below identify `com.google.zxing.qrcode.decoder`, which is used in this translation.

  - *ModVersion* parameterizes standard tables, including Reed–Solomon block and codeword counts, as described in Section 3.3.1.
  - *ModMode* adjusts Character Count Indicator lengths so the larger number of encoded characters can be represented.
  - *ModDecodedBitStreamParser* and *ModBitMatrixParser* allow the bit-plane parsers to obtain version and format information from the separate luminance matrix.
  - *ModDecoder* launches decoding, including error correction, and interleaves the data parsed from the separate bit matrices into the single sequence required by Reed–Solomon processing.

- `com.google.zxing.qrcode.fullcolor` is entirely new and groups components specifically concerned with chromatic image data:

  - *ColorBinarizer* maps sampled color values to bit sequences using the chromatic-distance metric introduced in Section 3.3.4.
  - *ColoredPixel*, *RgbPixel*, and *YCbCrPixel* represent pixel values in the two formats and support conversion.
  - *FullColorImageHandler* and its subclasses *YCbCrColorImageHandler*, *YCbCr420SpImageHandler*, and *RgbColorImageHandler* manage and manipulate scanned-image data.

4.5.3 Platform-Dependent Packages

The platform-dependent packages were changed only as needed to launch the new core components.

[9] The image itself is digital, but a pixel may assume millions of values; "analog" here emphasizes the conversion from that broad range to a short bit sequence.

- On Android, the digital camera image must retain color. The original project extracted luminance from raw YCbCr data and discarded subsampled red and blue chrominance because it was unnecessary. In `com.google.zxing.client.android`, *DecodeThread* is extended as *ModDecodeThread* so the thread supplying camera frames continuously feeds full color images to the core.

- Under Java Standard Edition, scanned color is not subsampled and pixels normally use RGB, simplifying processing. In `com.google.zxing.client.j2se`, a small adaptation of *GUIRunner* enables the new decoder. Future desktop work could apply additional image filters because desktop processors offer substantially more performance than mobile devices.

4.5.4 Backward-Compatibility Characteristics

The resulting application continues to detect and decode standard Quick Response codes. Initial work is common to standard and color QR decoding: finding position detection patterns, determining sample positions, and reading version and format information. Without that shared process, standard QR Codes could not be regarded as a subset of the high-capacity color symbology.

After common processing, the algorithm looks for the legends[10] introduced to generalize the selectable palette and correct chromatic distortion. If they are absent, standard processing continues; if present, the additional modules are launched. Extra computation is therefore incurred only when needed, with only a small overhead otherwise.

Observation. For a standard QR Code, the extended reader performs only the additional and unsuccessful legend search. A legend may be replicated on one row adjacent to a symbol containing from 21 to 177 rows, so processing that row is a modest overhead regardless of the exact future placement. Chapter 5 measures this overhead quantitatively.

4.5.5 Operational Summary of the Added Software Modules

Consider now the second scenario: decoding the new color symbology. Once the Detector[11] has found position detection patterns, estimated pixels per module,[12] determined the symbol dimension and probable version, recognized alignment patterns, and confirmed the legends, sampling can no longer return a single matrix in which each position is zero or one according to luminance. Each pixel selected to represent

[10] This new pattern generalizes the selectable color palettes and, more importantly, supports correction of chromatic distortion.

[11] Here "Detector" means the functions that identify and locate symbols in images and analyze their orientation. In ZXing, these procedures are concentrated in `com.google.zxing.qrcode.detector`.

[12] That quantity is expressed as the number of pixels per module in the particular scan being processed.

a module must be compared with legend colors using the chosen distance metric and assigned the corresponding multi-bit value.

The sampled chromatic data are still masked. Standard unmasking takes one bit matrix and XORs it with the mask pattern. The most direct and space-efficient solution is to store the color data as n bit matrices, where n is bits per module, together with one luminance matrix. Standard unmasking and bit-stream extraction can operate separately on each of the n matrices. Their resulting streams are then interleaved in the correct order to obtain one sequence for error correction. The luminance matrix supplies version and format information in the ordinary way.

For this to work, the generator must select the best mask from the color symbol's luminance matrix, mapping each light color to zero and each dark color to one as the standard does for black and white. If a light color corresponds to one bit sequence, complementing every bit of that sequence must identify a dark color, and vice versa. Without these conditions the process loses its effectiveness.

The interleaved bit stream can finally be passed to the Decoder,[13] meaning the components that decode extracted data rather than the image-processing functions that recover it. The Decoder interprets the bytes as Reed–Solomon blocks, corrects erroneous codewords, and continues decoding. Successful operation requires parameterizing the Version tables by `bits_per_module` and modifying Mode data for the Character Count Indicator, exactly as in the generator.

[13]Here "Decoder" means all software components that decode the extracted data, excluding the image-processing functions required to recover those data. In ZXing, these procedures are concentrated in `com.google.zxing.qrcode.decoder`.

Chapter 5

Experimentation and Analysis of Results

This chapter examines the performance of the color Quick Response reader after symbols are printed [7] on ordinary paper at several quality levels and scanned either by a conventional desktop scanner or by smartphone cameras in the mobile environment.

Having implemented prototype generator and reader applications, described in Chapter 4, we can assess the results. Desktop testing used an Epson DX6000 inkjet multifunction printer and scanner. Mobile testing installed the reader on HTC Tattoo and HTC Magic smartphones. Both ran Android 1.6, also known as Donut. Although the devices were similar in many respects, their results differed appreciably because the HTC Magic camera had autofocus while the HTC Tattoo camera did not.

The lack of autofocus may matter little in some ordinary photographs, but it is important when imaging two-dimensional codes or barcodes generally. Below a certain printed size, symbols become effectively unreadable on devices whose cameras do not provide autofocus.

5.1 Operational Examples

The preceding discussion described the process used to produce color Quick Response codes with increased information density but did not yet show an example. The following symbols were generated with the encoder prototype from Section 4.4.

Figure 5.1: Example of a type 2H code using a four-color scheme (two bits per module) to represent data optically.

Editorial note (2026): The caption was amended for the 2026 digital edition to remove the direct reference to "Quick Response." The image is unchanged and records output generated by the author's 2010 prototype.

Figure 5.2: Example of a type 6H code using a four-color scheme (two bits per module).

Editorial note (2026): The caption was amended for the 2026 digital edition to remove the direct reference to "Quick Response." The image is unchanged and records output generated by the author's 2010 prototype.

The next screenshots show successful readings of the preceding symbols from ordinary paper with the HTC Tattoo. This device captured only 320×240 pixels, lacked autofocus, and used the chromatic subsampling described in Section 4.3.3. The data in Figures 5.1 and 5.2 were therefore recovered from images occupying only 115,200 bytes: 320×240 pixels multiplied by $3/2$, because each pixel uses an average of one and a half bytes, or twelve bits.

Figure 5.3: HTC Tattoo reading of the symbol in Figure 5.1.

Editorial note (2026): This historical screenshot is unchanged and records the author's 2010 prototype and experiment.

Figure 5.4: HTC Tattoo application reading the symbol in Figure 5.2.

Editorial note (2026): This historical screenshot is unchanged and records the author's 2010 prototype and experiment.

Autofocus is important for small symbols. The next screenshot was obtained with the autofocus-equipped HTC Magic from a code printed at less than one square inch

and carrying nearly 400 bytes. Capture resolution was 320 × 480 pixels.

Figure 5.5: HTC Magic reading of a four-color Version 8L symbol measuring less than one square inch.

Editorial note (2026): This historical screenshot is unchanged and records the author's 2010 prototype and experiment.

These screenshots show the Android reader using smartphone cameras. Images from an ordinary scanner were considerably better than Camera Preview frames, partly because they lacked chromatic subsampling. They allowed denser four-color codes and some sixteen-color symbols to be decoded, although the reliability of the latter still required future work.

Figure 5.6: Successful desktop decoding of a sixteen-color symbol scanned with an Epson DX6000.

Editorial note (2026): This historical screenshot is unchanged and records the author's 2010 prototype and experiment.

5.2 Analysis of the Added Overhead

Performance tests measured the computational cost of introducing color and its associated processing. A meaningful quantitative comparison was possible by running the original ZXing reader and the extended color reader on the same machine. Because the second was based on ZXing and shared substantial code, the results could isolate the added work.

Dozens of different symbols were considered. Each was decoded twenty times and the mean execution time used as its reference. Absolute times depend on the hardware and are meaningful only for runs on the same machine, performed close together to reduce differences in system load.

All reported tests ran on a desktop machine with a 1.73 GHz Intel dual-core processor, 3 GB of RAM, Slackware Linux 13.0, and a light load. A dedicated test application executed the runs consecutively and calculated the means. Each decoding interval, in milliseconds, began after the scanned image had been loaded into memory, immediately before processing, and ended when the encoded message had been

recovered.

Standard QR Codes are denoted by the abbreviation “QR.”

Editorial note (2026): The remainder of the corresponding 2010 sentence used a provisional name, omitted in this edition. In the comparison, “4-color symbols” and “16-color symbols” identify cases of the new symbology, called HCC2D in publications subsequent to the thesis [4]. The technical meaning is unchanged.

Version	QR	4-color symbols	16-color symbols
1L	122	132 (7.57%)	135 (9.62%)
1M	123	136 (9.55%)	138 (10.87%)
1Q	129	133 (3.01%)	135 (4.45%)
1H	131	135 (2.97%)	136 (3.68%)
5L	143	171 (16.38%)	206 (30.59%)
5M	151	174 (13.21%)	208 (27.40%)
5Q	163	181 (9.95%)	208 (21.63%)
5H	161	182 (11.54%)	209 (22.97%)
10L	176	209 (15.79%)	246 (28.45%)
10M	188	208 (9.61%)	249 (24.49%)
10Q	189	210 (10.0%)	273 (30.77%)
10H	190	212 (10.37%)	282 (32.62%)
20L	240	349 (31.23%)	387 (37.98%)
20M	253	334 (24.25%)	398 (36.43%)
20Q	250	337 (25.81%)	389 (35.73%)
20H	257	323 (20.43%)	395 (34.93%)
30L	333	447 (25.50%)	474 (29.75%)
30M	350	430 (18.60%)	460 (23.91%)
30Q	347	437 (20.59%)	478 (27.40%)
30H	338	416 (18.75%)	506 (33.20%)
40L	430	483 (10.97%)	550 (21.81%)
40M	415	481 (13.72%)	540 (23.15%)
40Q	420	500 (16.0%)	566 (25.79%)
40H	373	485 (23.09%)	552 (32.42%)

Figure 5.7: Mean decoding times in milliseconds for the different code categories.

Editorial note (2026): The experimental values are unchanged from the 2010 figure. The table was newly typeset for this edition, and its provisional color-format headings were replaced by neutral descriptions.

In the color QR columns, the value in parentheses beside the twenty-run mean is the additional percentage of time required relative to a standard QR Code of the same version and error-correction level.

5.3 Code Reliability

Having described the processing overhead introduced by chromatic information, we now consider reliability. Tests covered dozens of color codes printed on ordinary paper with an Epson DX6000 at a total size of one square inch, using several alternative palettes.

An ordinary Epson DX6000 printer/scanner could read four-color Quick Response codes carrying more than one kilobyte, printed at one square inch and scanned at 600 dpi, using only level-L error correction at 7%. Symbols with capacities slightly below

one kilobyte, again printed at one square inch, could be decoded successfully even at low print quality other than draft mode and at a scan resolution of only 300 dpi.

Chapter 6

Conclusions and Future Work

This thesis introduced a new code symbology based on Quick Response codes that uses chromatic information to increase information density. Generator and decoder applications were implemented, and the technology was tested in both desktop and mobile environments.

The proposed specifications proved viable and the resulting codes readable. Tests on desktop and mobile platforms demonstrated generation and reading while accounting for symbol printing and scanning. Innovation was driven by the widespread availability of mobile devices capable of acting as optical readers through an integrated camera and suitable application. In view of the present and especially future capabilities of smartphone cameras, it was worthwhile to exploit more fully the chromatic information they can capture.

This direction also differed from Microsoft's High Capacity Color Barcode, a color two-dimensional symbology like the one proposed here but a proprietary solution without characteristics sufficiently strong to make it superior to existing two-dimensional codes. The approach adopted in this thesis instead introduces color into an effective and mature symbology, Quick Response, whose specification is fully published in ISO/IEC 18004.

The work addressed substantial challenges in image processing and computer vision, Android mobile programming, reuse of poorly documented open-source projects, and the conceptual redesign of Quick Response specifications that formed the foundation for the implementation.

Although the prototype's final performance was satisfactory, several avenues for future work remain. First, new chromatic-distance metrics should be defined and tested to identify those that best evaluate similarities and differences among colors perceived by the reader. This could reduce module-classification errors and improve overall reliability.

Image-processing optimizations could improve symbol detection and module sampling, and filters could be applied to the digital image. A Gaussian low-pass filter, for example, blurs the image and removes erroneous pixels whose colors differ sharply from their neighbors. Because many filters are available, future work could determine

which most improves readability and whether several should be cascaded when the device has sufficient computing power.

The placement of chromatic legends, currently replicated along the sides of the code, could also be refined. Other locations may provide better color sampling. One initial idea is to replicate legends on at least two adjacent sides to correct possible misalignment along both horizontal and vertical axes.

Palette selection also deserves careful study. Although the generator should remain free to select colors, some palettes will perform much better than others, and identifying the most effective choices would be valuable.

Data masking offers another avenue for improving reliability. The reworked process produced promising results, but alternatives could be evaluated under different lighting conditions, including artificial light, sunlight, and dim environments. Such conditions may be more critical for color codes than for traditional black-and-white symbols and therefore merit detailed analysis.

Finally, Android's interaction with the camera could be reconsidered. Camera Preview conveniently supplies consecutive captures without burdening the user, but its chromatic information is poor and subsampled, as discussed throughout the thesis.

List of Figures

1.1 An example of an EAN-8 barcode. 6
1.2 Comparison of a 1D code (barcode) and a 2D code (here a QR Code). 7
1.3 Comparison of the principal two-dimensional code families and their maximum capacities. 8
1.4 An example of a Microsoft HCCB code: Microsoft Tag. 9
1.5 Structure of a PDF417 code. 10
1.6 Example of a Codablock F code. 10
1.7 Structure of a Data Matrix code. 11
1.8 Four- and eight-color Microsoft HCCB codes. 13

3.1 Conceptual relationships among the capacity parameters discussed in the 2010 text. 20
3.2 Conceptual dependencies among version, error correction, data size, and input capacity. 22
3.3 Conceptual relationships used to extend the error-correction parameters. 23
3.4 Conceptual extension logic for the Character Count Indicator. 24
3.5 Author-created schematic of the functional regions of a Model 1 symbol. 27
3.6 Example of data masking under the Quick Response standard. 28
3.7 Example of data masking reworked for the new symbology. 29
3.8 Alternative arrangements of colored modules within the legends. Actual colors are replaced by their corresponding luminance values. . . . 29

5.1 Example of a type 2H code using a four-color scheme (two bits per module) to represent data optically. 44
5.2 Example of a type 6H code using a four-color scheme (two bits per module). 44
5.3 HTC Tattoo reading of the symbol in Figure 5.1. 45
5.4 HTC Tattoo application reading the symbol in Figure 5.2. 45
5.5 HTC Magic reading of a four-color Version 8L symbol measuring less than one square inch. 46
5.6 Successful desktop decoding of a sixteen-color symbol scanned with an Epson DX6000. 47
5.7 Mean decoding times in milliseconds for the different code categories. 48

Bibliography

[1] Microsoft Research, *High Capacity Color Barcodes (HCCB)*, `http://research.microsoft.com/en-us/projects/hccb/`, May 2010.

[2] D. Parikh and G. Jancke, *Localization and Segmentation of a 2D High Capacity Color Barcode*, pp. 1–6, January 2008.

[3] International Organization for Standardization, *ISO/IEC 18004:2006, QR Code bar code symbology specification*, `http://www.iso.org`, 2006.

[4] A. Grillo, A. Lentini, M. Querini, and G. F. Italiano, *High Capacity Colored Two Dimensional Codes*, Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT 2010), pp. 709–716, 2010. doi:10.1109/IMCSIT.2010.5679869.

Editorial note (2026): This entry replaces the provisional title and preliminary conference details printed in the 2010 thesis with the final published citation; see "About This Digital Edition."

[5] K. Fukuchi, *Libqrencode, a C library for encoding data in a QR Code symbol*, `http://megaui.net/fukuchi/works/qrencode/`, 2010.

[6] ZXing Team, *ZXing, an open-source, multi-format 1D/2D barcode image processing library*, `http://code.google.com/p/zxing/`, 2010.

[7] Denso Wave, *Quick Response Code – printer head density and module size*, `http://www.denso-wave.com/qrcode/qrgene3-e.html`, 2000.

Informazioni su questa edizione digitale

Tesi originale **Anno**
Analisi e progettazione di codici bidimensionali ad alta capacità. **accademico**
Sviluppo del lettore per gli ambienti desktop e mobile. 2009/2010

Contesto accademico originale. Questa edizione bilingue, versione 1.0, presenta la tesi di laurea specialistica in lingua italiana di Marco Querini, discussa il 23 luglio 2010 presso il Corso di laurea in Ingegneria Informatica, Facoltà di Ingegneria, Università degli Studi di Roma Tor Vergata.
Correzione bibliografica. Alla discussione del luglio 2010, il primo contributo congressuale sulla nuova simbologia non era ancora uscito; la bibliografia riportava quindi il titolo provvisorio *High Capacity Colored QR Codes* e dati congressuali preliminari. Questa edizione riporta la citazione definitiva registrata dall'editore, il cui titolo neutrale evita anche di suggerire che la simbologia a colori proposta sia essa stessa un QR Code: *High Capacity Colored Two Dimensional Codes*, IMCSIT 2010, pp. 709–716, doi:10.1109/IMCSIT.2010.5679869.
Marchio e ambito. "QR Code" è un marchio registrato di DENSO WAVE INCORPORATED in Giappone e in altri Paesi. Il capitolo intitolato *Quick Response a Colori ad Alta Capacità* esamina come introdurre il colore nella struttura QR Code per aumentarne ulteriormente la capacità. Il titolo usa "QR Code" soltanto per identificare l'oggetto dell'indagine accademica; non implica che il capitolo o la simbologia—denominata HCC2D nelle pubblicazioni successive alla tesi—siano sponsorizzati, approvati o affiliati a DENSO WAVE INCORPORATED.

Il volume è articolato in due parti:

1. una traduzione inglese, presentata per prima per facilitarne l'accesso internazionale;
2. *Italiano — testo originale del 2010*, che conserva la sequenza e il testo italiani, salvo gli interventi editoriali qui dichiarati.

La tesi italiana originale del 2010, conservata dall'autore e dall'Università, è la fonte autorevole e prevale in caso di discrepanza, ambiguità o differenza interpretativa. La traduzione ne segue terminologia e contesto storico senza aggiornare implicitamente le affermazioni tecniche alla luce di standard o prodotti successivi.

In ciascuna sequenza linguistica, una nota editoriale registra l'omissione prudenziale del Capitolo 2 originale, evitando di riproporre gli elementi dettagliati della specifica QR Code.

La Figura 1.3 è una nuova tavola vettoriale con schemi indipendenti e capacità massime pubblicate. Le tabelle derivate dallo standard delle Figure 3.1–3.4 non sono riprodotte: schemi concettuali originali ne conservano solo relazioni e logica di calcolo. Le altre figure e tabelle prima del Capitolo 4 sono ricostruzioni da sorgenti modificabili. Le Figure 5.1, 5.2 e 5.7 recano interventi dichiarati; la 5.7 conserva i valori sperimentali originali, con intestazioni neutrali e una nota blu adiacente che registrano il nome HCC2D adottato successivamente. Dal Capitolo 4 in poi, le altre figure restano testimonianze storiche della ricerca originale.

Convenzioni editoriali. Le note del 2026 sono in blu scuro. Il testo discorsivo italiano conservato in nero segue quello del 2010, con la correzione di un solo refuso tipografico; ogni omissione o sostituzione è dichiarata da una nota blu adiacente.

Risorse correlate:

- *Specifica tecnica.* La successiva simbologia HCC2D è documentata nella specifica pubblica *Specifica del codice HCC2D*, versione 0.9.0.
- *Sito ufficiale.* hcc2d.com.

Il mio lavoro di tesi è dedicato a coloro che mi sono stati vicini in questi anni di studi, in particolar modo ai miei genitori e a mia sorella.

Indice

Introduzione **1**

1 Codici Bidimensionali: Studio ed Analisi **8**

1.1 I Codici a Barre 8

1.1.1 Una Semplice Definizione 9

1.1.2 Da Analogico a Digitale 10

1.1.3 Evoluzione dei Codici a Barre 11

1.2 I Codici Bidimensionali 12

1.2.1 Ambiguità della Definizione 12

1.2.2 Tipologie di Codici Bidimensionali 12

1.2.3 Codici 2d a Barre Impilate: PDF417, Codablock 15

1.2.4 Codici 2d a Matrice 17

1.2.5 Codici 2d a colori: Microsoft HCCB 20

2 Il Codice Quick Response (ISO/IEC 18004) **23**

3 Quick Response a Colori ad Alta Capacità **24**

3.1 Obiettivi del Progetto 25

3.1.1 Project Scope 26

3.2 Problematiche Aperte 27

3.2.1 Osservazioni sui Function Patterns 27
3.2.2 Osservazioni sull'Area di Codifica 28
3.2.3 Principali Problematiche e Sfide da Affrontare 30
3.3 Specifiche di Progetto Proposte 33
3.3.1 Estensione delle Tabelle dei Codici Quick Response 34
3.3.2 Gestire l'Introduzione dei Colori nei Quick Response 45
3.3.3 Reinventare il concetto di Data Masking 49
3.3.4 Definire una Metrica di Distanza Cromatica 55

4 Progetto e Realizzazione di Generatore e Lettore 57

4.1 I Progetti di Riferimento 58
4.2 Introduzione ad Android O.S. 59
4.3 Impatto di Android O.S. 61
4.3.1 Limitazioni di Portabilità del Codice 61
4.3.2 Un Emulatore Carente per il Progetto 62
4.3.3 Uso della Fotocamera: Rischi per il Progetto e Considerazioni sullo Sviluppo 62
4.4 Realizzazione del Generatore 68
4.4.1 Struttura dei Sorgenti 69
4.5 Realizzazione del Lettore 75
4.5.1 Struttura del Progetto 76
4.5.2 Packages appartenenti al "Core": Indipendenti dall'Architettura 78
4.5.3 Packages Dipendenti dalla Piattaforma 83
4.5.4 Caratteristiche di Retro-Compatibilità 84
4.5.5 Sintesi di Funzionamento dei Moduli Software Introdotti . . . 85

5 Sperimentazione ed Analisi dei Risultati **89**

5.1 Esempi di Funzionamento 90

5.2 Analisi dell'Overhead Introdotto 94

5.3 Affidabilità dei Codici 97

6 Conclusioni e Sviluppi Futuri **98**

Elenco delle figure **102**

Bibliografia **103**

Introduzione

Al giorno d'oggi i codici a barre sono principalmente diffusi come strumenti di identificazione dei prodotti in commercio; dal momento che sono presenti su quasi tutti gli articoli in vendita, la funzione che svolgono nei negozi è comunemente nota, tuttavia si intravedono potenzialità legate a nuovi possibili usi. Si tratta di oggetti costituiti da elementi grafici a contrasto che esprimono una rappresentazione ottica dei dati interpretabile dalle macchine (machine-readable), tramite utilizzo di lettori ottici; sono strumenti capaci di rendere persino la carta stampata un mezzo digitale. La lettura dei codici a barre rappresenta quindi un processo di recupero automatico delle informazioni, caratterizzato da alte prestazioni relativamente ad affidabilità, accuratezza e velocità di decodifica.

La necessità d'innovazione dei codici a barre è derivata da quell'unico difetto, tuttavia critico, rappresentato dal fatto di avere una capacità di memorizzazione estremamente contenuta, in conseguenza di una densità di informazione di pochi caratteri per pollice quadrato. Non vi è alcun dubbio che per l'utilizzo che spesso se ne fa, che di solito prevede la rappresentazione del semplice identificativo di un determinato prodotto, il classico codice a barre è quanto di più adatto, caratterizzato com'è da una struttura monodimensionale che ne rende la lettura un processo estremamente rapido ed affidabile; tuttavia, per sfruttare le potenzialità latenti di questi oggetti e supportare nuovi casi d'uso, è stato necessario introdurre una struttura bidimensionale. Uno

scenario supportato dalla nuova tecnologia è dato ad esempio dalla possibilità di rappresentare direttamente, anziché il solo identificativo del prodotto, tutte le informazioni di interesse a lui corrispondenti (produttore, modello, eventuali specifiche tecniche, ...). Nascono così i cosiddetti codici 2D, che sfruttano entrambe le dimensioni per la codifica dei dati e sono caratterizzati da una struttura a barre o a matrice, sebbene, a causa di un abuso di notazione, in entrambi i casi vengano comunemente indicati con l'appellativo di codici a barre.

Nonostante le alte capacità ottenute con la tecnologia 2D, in queste simbologie esiste ancora un elemento vincolante che ne limita le prestazioni: poiché gli elementi a contrasto hanno sempre due sole possibili gradazioni, bianche o nere, ogni singola cella di un codice a matrice non potrà in ogni caso portare più di un bit di informazione.

Questo lavoro di tesi si colloca nel contesto dell'innovazione dei codici bidimensionali, nella direzione dell'incremento della densità di informazione tramite l'introduzione del colore, fatto che consente quindi alla singola cella (o modulo) del codice di rappresentare graficamente oltre un bit di informazione. In particolare, con riferimento a quanto sopra accennato, l'esigenza di incrementare la capacità dei codici non è tanto relativa ad un tentativo di sostituire i classici codici a barre con una nuova tipologia negli utilizzi già comuni, bensì di produrre simboli che supportino nuove funzionalità, grazie ad una maggiore densità di informazione.

In questo senso, la più recente spinta all'innovazione dei codici è arrivata dalla grande diffusione di quei dispositivi mobili, come i moderni smartphones, capaci di lavorare come lettori ottici previa installazione dell'opportuna applicazione, facendo uso della fotocamera integrata. Si intravede una possibilità nuova, per cui si può ipotizzare uno scenario futuro in cui chiunque, avendo con se il proprio telefono cellulare, ha a disposizione un potenziale lettore ottico di codici 2D. Lo scenario di riferimento

si modifica in tal senso, passando, da casi d'uso relativi ad una limitata fascia di utenti che hanno a disposizione hardware special purpose per un utilizzo tipicamente lavorativo, a casi d'uso che coinvolgono la grande massa di consumatori in scenari completamente nuovi.

Appare evidente, quindi, che la capacità di rappresentare un maggior quantitativo di dati nel medesimo spazio, che sottintende la necessità di produrre simboli a maggiore densità, rappresenta una delle sfide principali per la prossima generazione di codici a barre.

Il contributo di questo lavoro è lo sviluppo di una nuova tecnologia di simboli ad alta capacità, a partire da una base strutturale ripresa dai codici Quick Response (ISO/IEC 18004), di cui si apprezzano caratteristiche come la rapidità di lettura e l'affidabilità, quest'ultima ottenuta grazie ad un'ottima gestione delle problematiche di distorsione, disallineamento e correzione degli errori; infatti, la lettura di codici pone sfide per nulla banali in relazione a processamento delle immagini e computer vision. Un'immagine scansionata può ritrarre un simbolo in maniera anche molto diversa dall'originale: se pensiamo al fatto che non si possono fare assunzioni sull'orientazione del codice per come viene percepito dal lettore; se consideriamo che un simbolo, ad esempio quadrato, a causa di uno scatto che ne fa una proiezione lungo una dimensione, può deformarsi in un rettangolo o in un quadrilatero generico; se si ragiona sul fatto che non si può prescindere dall'avere una garanzia di funzionamento anche relativamente a quelle situazioni in cui i codici sono parzialmente danneggiati o in quei casi in cui la luminosità dell'ambiente non è ottimale. In questo contesto, si vogliono affrontare nuove sfide, in aggiunta a quelle a cui si è appena accennato, focalizzate alla gestione dell'introduzione del colore, che pone quantomeno problematiche di rilevazione e correzione delle distorsioni cromatiche relative. Non solo, in aggiunta alle problematiche

di image processing, va gestito il rimodellamento di un oggetto complesso come può essere il codice Quick Response, in maniera tale che l'atto di riprogettare le specifiche corrispondenti non condizioni l'affidabilità dei simboli, con un trade-off tra incremento della capacità e leggibilità dei codici da rispettare. Superati questi scogli concettuali, vengono affrontate le problematiche di realizzazione del generatore e del lettore per la simbologia di codici proposta, dovendo ottenere delle prestazioni accettabili anche in quei casi in cui il dispositivo mobile produca scatti di bassa qualità, caratterizzata da un sottocampionamento delle informazioni cromatiche che vede ogni pixel rappresentato, in media, da soli 12 bits, di cui 8 unicamente per la luminosità. Vengono considerate inoltre le problematiche relative ai processi di stampa su carta comune e le eventuali difficoltà pratiche legate al fatto di non avere possibilità di produrre stampe a colori. In questo senso la simbologia proposta non vincola lo schema di colori effettivo da usare, che rimane personalizzabile in base alle effettive esigenze. Soluzioni pratiche vedono l'uso di codici con schemi a 4 gradazioni di grigi per stampanti in bianco e nero, o l'uso di colori primari come ciano e magenta qualora le stampanti siano imprecise nel produrre colori compositi, tuttavia le prestazioni risentono delle scelte prese e non tutti gli schemi di colori possibili producono risultati accettabili. Ci si aspetta che le prestazioni del lettore prodotto, in ambiente desktop, siano superiori a quelle ottenute in ambito mobile, vista la migliore qualità della scansione, tuttavia si intravedono le potenzialità latenti, relative al prossimo futuro, di tutti quei dispositivi smartphones il cui hardware ottico in dotazione è di anno in anno più performante; si potrebbe facilmente ipotizzare, per gli anni a venire, di vedere fotocamere con flash, autofocus e ad alte risoluzioni sulla maggior parte dei dispositivi in mano agli utenti, che seppure già abbastanza diffusi, non coprono ancora l'intera fascia dei consumatori, essendoci ancora una percentuale non trascurabile di persone legate a

telefoni della precedente generazione, i quali andranno però lentamente a scomparire. La caratteristica di originalità di questo lavoro non risiede, ad ogni modo, nell'utilizzare gli smartphones per la lettura dei codici, dal momento che lettori di codici a barre sono già disponibili tanto per dispositivi con sistema operativo Android, quanto per gli iPhones; tuttavia, il fatto di prendere atto delle potenzialità di questi dispositivi ha portato all'idea di riprogettare dei codici come i Quick Response che, seppure ancora meritatamente attuali, all'epoca[1] della loro ideazione non potevamo tenere in considerazione né la futura nascita di dispositivi come gli smartphones, né la grande diffusione delle stampanti a colori a basso costo. L'idea di produrre codici a colori ad alta capacità è quindi estremamente attuale se si pensa che la stessa Microsoft ultimamente ha realizzato una tecnologia chiamata "High Capacity Color Barcode", che introduce appunto il colore nei simboli. La soluzione di Microsoft presenta però lo svantaggio di essere proprietaria, inoltre la struttura progettata per i propri codici non è migliore di quelle adottate da simbologie come Quick Response o Data Matrix; ecco perché il rimodellamento di una tecnologia già matura come quella relativa ai "QR", per supportare l'introduzione della componente cromatica, può conseguire potenzialmente un risultato che sia significativo.

Struttura della Relazione

Nel primo capitolo si analizza lo stato dell'arte dei codici bidimensionali, a barre o a matrice, a seguito di un paragrafo introduttivo sui classici codici a barre monodimensionali, avente lo scopo di introdurre l'argomento. Il fine ultimo è quello di permettere la comprensione del quadro relativo alle tecnologie attualmente in uso, da usare come punto di partenza per capire quale struttura abbia senso rimodellare per introdurre

[1] si ricorda che si tratta degli anni '90, in cui esisteva a malapena la prima generazione di telefoni cellulari.

una simbologia a colori. Esistono infatti numerose varietà di codici bidimensionali, con caratteristiche a volte simili, ma con strutture diverse, che possono avere o meno punti di forza in specifiche caratteristiche. La scelta del codice da rielaborare è caduta sul Quick Response, avente proprietà di elevata capacità, affidabilità e velocità di decodifica, nonostante non si possa dire che una tecnologia come Data Matrix sia da meno. L'atto di scegliere quale codice rielaborare e riprogettare per generare un simbolo ad alta capacità che sfruttasse l'informazione cromatica non poteva infatti non prendere in considerazione le qualità dei vari tipi di codici.

Nel secondo capitolo, quindi, si scende in dettaglio sulle caratteristiche strutturali proprie di un simbolo Quick Response, perché, ovviamente, si deve fornire un quadro di riferimento sulle funzionalità offerte da ogni singolo componente interno prima di poter fare ragionamenti su quali specifiche rimodellare e in quale maniera.

Nel terzo capitolo si vanno appunto a definire le nuove specifiche, selezionando attentamente cosa ereditare e cosa rielaborare, ma soprattutto stando attenti ad evitare di andare incontro al rischio, non trascurabile, di produrre un codice che abbia delle incongruenze interne, che degradino le prestazioni o ne compromettano l'affidabilità. Si affrontano quindi le problematiche relative ad un image processing che tenga conto della gestione cromatica, si manipolano le tabelle interne alle specifiche dello standard, si cambia la maniera di spalmare i dati sulla superficie del simbolo, si reinventa la stessa procedura di Data Masking, si cerca di definire una metrica di distanza cromatica.

Nel quarto capitolo si descrive come è stata condotta la realizzazione di un generatore e di un lettore per la nuova simbologia di codici, a testimonianza che le specifiche rielaborate mantengono la proprietà di essere coerenti, essendo alla base di simboli che sono stati sperimentati effettivamente leggibili. Gli applicativi prodotti lavorano

sia in ambiente desktop, sia in ambiente mobile; in quest'ultimo caso relativamente a tutti quei dispositivi che montano il sistema operativo Android O.S., fabbricati da diversi produttori hardware, come possono essere Htc, Samsung, Lg o Motorola.

Il quinto e ultimo capitolo presenta alcuni esempi di generazione di codici Quick Response ad alta capacità, nonché prove di funzionamento del lettore su dispositivi mobili come Htc Tattoo e Htc Magic. Vengono proposte inoltre misurazioni quantitative relative all'overhead introdotto in termini di costi computazionali, vista la necessità di moduli software aggiuntivi per la gestione delle problematiche cromatiche; a tale proposito sono stati condotti esperimenti su decine di istanze della nuova simbologia di codici.

Infine si è cercato di capire qual'era la massima densità di informazione raggiungibile, tale che un codice prodotto e stampato su carta comune, tramite una stampante non professionale, potesse essere ancora leggibile. A questo proposito si sono fatti dei test di stampa tali da produrre codici di una dimensione prestabilita al fine di sperimentare quale capacità complessiva effettivamente utilizzabile nella pratica, potesse avere un codice appartenente alla nuova simbologia.

Capitolo 1

Codici Bidimensionali: Studio ed Analisi

In questo capitolo si vuole mettere a fuoco lo stato dell'arte dei codici bidimensionali, con un'attenzione particolare allo standard Quick Response ("QR"). Al fine di comprendere appieno il progetto di codici Quick Response ad Alta Capacità (obiettivo centrale del lavoro, insieme alla realizzazione del lettore ottico corrispondente), è infatti necessario, in primis, fare il punto della situazione relativamente allo stato attuale. Per avere una visione completa, si introdurranno inizialmente i classici codici a barre, che pur non essendo bidimensionali, costituiscono l'origine della tecnologia di interesse; essendo inoltre codici di uso comune nel mondo moderno, si consentirà anche ad un lettore non esperto di essere introdotto agevolmente alla comprensione del capitolo. Seguirà una sezione dedicata a quelle classi e tipologie di codici bidimensionali attualmente predominanti, al fine di introdurre lo standard ISO/IEC 18004 (relativo ai codici Quick Response) che verrà trattato nei capitoli successivi.

1.1 I Codici a Barre

Chiunque al giorno d'oggi, ha un'idea di cosa sia un codice a barre, poiché il suo utilizzo è diventato parte integrante della vita di tutti i giorni. Non tutti sarebbero ovviamente in grado di spiegare il procedimento di lettura ottica che li vede coinvolti, ma è impossibile non essersi mai imbattuti in una cassiera che tenta di scansionare il

codice posto sul prodotto che si sta acquistando. Questo è soltanto uno dei tanti casi d'uso, anche se probabilmente uno tra i più comuni; meno ovvio è probabilmente l'utilizzo dei codici a barre per permettere l'identificazione dei convogli in transito lungo le linee ferroviarie. È una di quelle tecnologie pervasive, tendenti cioè a diffondersi ovunque, ed il cui utilizzo è divenuto così d'uso comune, che non si fa quasi più caso di averla sotto gli occhi. Su quanti documenti infatti, come fatture o biglietti di vario tipo, pur non attirando la nostra attenzione, sono presenti i codici a barre.

Cosa ci offre quindi l'utilizzo dei codice a barre? Da un punto di vista delle funzionalità svolte, esso consente l'automazione di alcuni processi, come l'identificazione degli oggetti su cui i codici sono apposti, fatto a cui si è già accennato.

1.1.1 Una Semplice Definizione

La maniera più semplice di definire i codici a barre è di dire che sono un insieme di elementi grafici a contrasto disposti in modo da poter essere letti da un sensore ottico. L'immagine scansionata del codice, così come è stata percepita dal sensore (quindi non necessariamente equivalente al codice originale per la presenza di eventuali distorsioni), va decodificata per restituire l'informazione contenuta in essa.

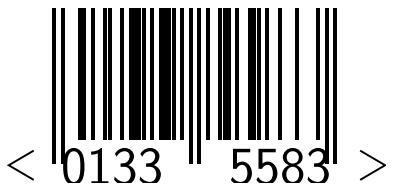

Figura 1.1: Un esempio di Codice a Barre Ean-8.
Nota editoriale (2026): Questo esempio EAN-8 è stato generato indipendentemente dall'autore per l'edizione digitale e non riproduce il campione raster del 2010.

Un codice a barre costituisce più precisamente una rappresentazione ottica di dati, interpretabile da una macchina (formato machine-readable) che abbia un supporto ottico: un foglio di carta che abbia impresso un codice a barre diviene un mezzo digitale,

poco diverso concettualmente da un supporto compact disk; quello che cambia è la densità di informazione e quindi la capacità di memorizzazione, che nei codici a barre è molto bassa.

1.1.2 Da Analogico a Digitale

È interessante notare come il procedimento di lettura di un generico codice riprenda il concetto di conversione analogico/digitale, riproponendolo nell'elaborazione di scansioni: a titolo di esempio, la scarsa o elevata luminosità, percepita dal sensore ottico relativamente a una determinata area di codice, potrebbe essere tradotta in linguaggio binario se la si valutasse in relazione ad una certa soglia (bassa luminosità = 1 e alta luminosità = 0, o viceversa). Questo concetto verrà poi ripreso in maniera estesa in relazione ai codici bidimensionali, poiché invece i codici a barre, in forza della loro simbologia monodimensionale, codificano dati soltanto in relazione a spessori e distanze tra linee parallele. Il principio sottostante rimane quello di cercare di estrarre dati digitali da un input caratterizzato da valori analogici: spessori e distanze tra linee parallele percepite dal sensore non potranno essere equivalenti alle grandezze del codice originale e andranno valutate per tentare di ricostruire spessori e distanze corrette, e da queste l'informazione memorizzata. Si noti ad ogni modo come anche la sola identificazione di oggetti di una certa forma in un'immagine, ottenuta da una scansione, sia un problema non banale: anche se questo tema verrà ripreso in seguito, vale la pena sottolineare da subito il fatto che la scansione del codice percepita dal lettore ottico non è esente da distorsioni di varia natura.

1.1.3 Evoluzione dei Codici a Barre

Il punto debole dei codici a barre è probabilmente la bassa densità di informazione, che permette soltanto la memorizzazione di pochi caratteri in una area che tipicamente prende alcuni centimetri quadrati. Prendiamo in considerazione l'altezza del codice di figura 1.1: anche se la incrementassimo non aumenteremmo la capacità di contenere informazioni, in quanto solo la componente orizzontale codifica dati (in base a spessori e distanze tra linee parallele), mentre la componente verticale viene mantenuta elevata anziché essere ridotta ad esempio ad un solo millimetro, solo per facilitare la lettura del codice (anche un utente frettoloso e " sbadato " nell' utilizzo del lettore ottico riesce a recuperare i dati qualora le barre siano sufficientemente alte; al contrario un codice che abbia barre di un millimetro richiede una mano molto ferma e precisa).

Per l'ottima affidabilità che li caratterizza, i codici a barre si sono diffusi ovunque, tuttavia il pesante vincolo sulla capacità di memorizzazione, ne limita fortemente i possibili nuovi utilizzi (l'intero codice di figura 1.1 codifica ad esempio solo 8 caratteri): per questo motivo si è sentita la necessità di progettare nuovi codici, strutturalmente diversi dai codici a barre, che ne rappresentano l'evoluzione; questi codici abbandonano la struttura monodimensionale dei codici a barre e sfruttano appieno entrambe le dimensioni per codificare dati, pur mantenendo una sufficiente affidabilità.

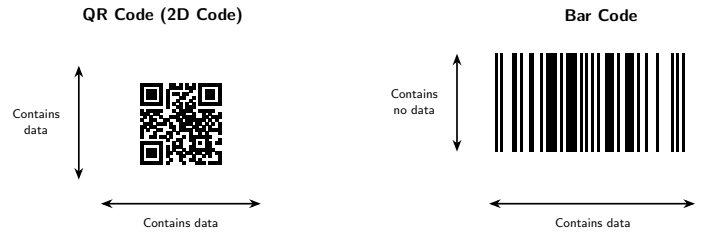

Figura 1.2: Confronto tra un codice 1D (a barre) e un codice 2D (qui di tipo QR)
Nota editoriale (2026): Questo confronto è uno schema originale creato indipendentemente dall'autore per l'edizione digitale; non riproduce i campioni grafici della figura del 2010.

1.2 I Codici Bidimensionali

In questa sezione si discuteranno i principali tipi di codice bidimensionali, per scendere poi profondamente in dettaglio in relazione ad un tipo specifico: i codici Quick Response. La comprensione dello standard che li definisce, è strettamente necessaria come base per capire modifiche ed estensioni che sono state progettate e realizzate per produrre dei codici Quick Response ad alta capacità, scopo di questo lavoro lavoro di tesi.

1.2.1 Ambiguità della Definizione

I codici bidimensionali, sono codici che riescono a sfruttare entrambe le dimensioni per la codifica dei dati, massimizzando così la capacità; impropriamente in alcuni testi vengono chiamati codici a barre bidimensionali, anche nei casi in cui non posseggano alcuna barra; nel seguito della trattazione si cercherà di evitare per quanto possibile di fare confusione, ad ogni modo per chiarire il concetto: i codici monodimensionali sono codici a barre, mentre per quanto riguarda i codici bidimensionali, esistono sia codici bidimensionali a barre che " a matrice ". I codici bidimensionali a barre sono anche noti come codici " a barre impilate ", composti in sostanza da vari codici a barre monodimensionali giustapposti, tanto che possono essere decodificati dai lettori tradizionali; i codici bidimensionali a matrice abbandonano invece del tutto ogni tentativo di rifarsi alla tecnologia precedente, divenendo strutturati secondo una scacchiera regolare.

1.2.2 Tipologie di Codici Bidimensionali

Indipendentemente dalle diverse varietà di codici bidimensionali che sono stati prodotti negli ultimi anni, le quali possono dare luogo anche a sensibili differenze prestazion-

ali, tutti i codici bidimensionali riescono a codificare rappresentazioni ottiche di una quantità di dati fortemente superiore rispetto a quanto avviene coi codici a barre (monodimensionali): si parla anche di alcuni kilobytes di contenuto, rispetto ai pochi bytes dei codici monodimensionali. Distinti i codici bidimensionali nelle due categorie principali, ovvero " a fasce impilate " e " a matrice ", si ha una grande varietà di codici per entrambe le classi: tra gli appartenenti alla prima tipologia abbiamo PDF417, Codablock e Supercode, mentre MaxiCode, Datamatrix e Quick Response sono esempi di codici a matrice.

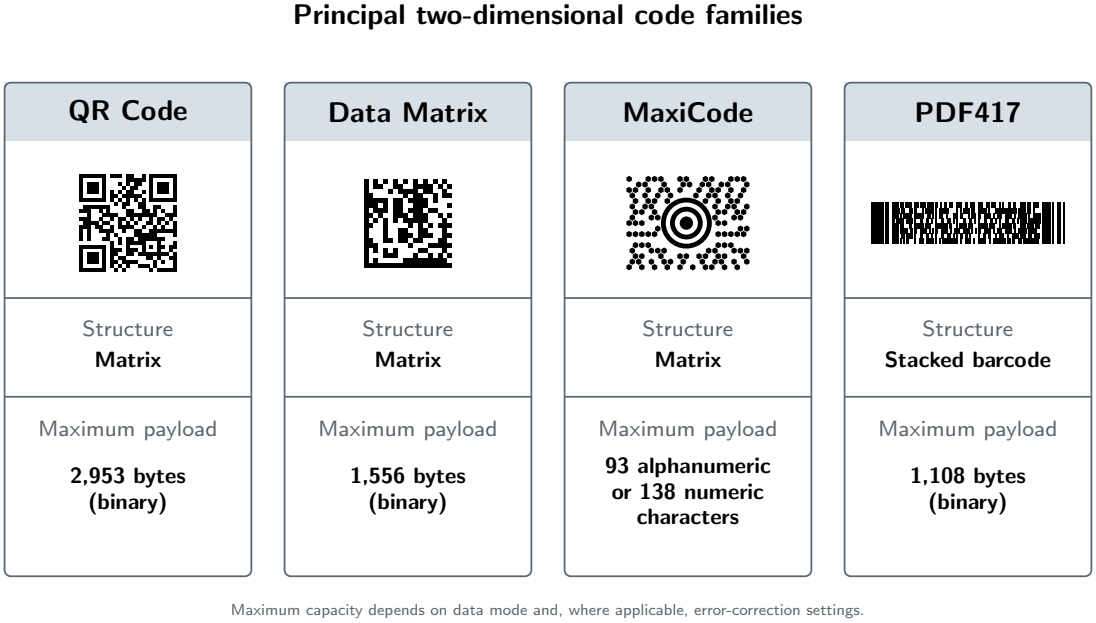

Figura 1.3: Confronto tra le principali famiglie di codici bidimensionali e le rispettive capacità massime.
Nota editoriale (2026): Questa nuova tavola vettoriale e i piccoli schemi dei simboli sono stati creati indipendentemente dall'autore per l'edizione digitale; non riproducono immagini degli standard. Le miniature QR Code, Data Matrix e PDF417 codificano "HCC2D digital edition"; MaxiCode è uno schema strutturale. I valori riportati sono capacità massime pubblicate. Per MaxiCode, che non presenta un limite binario direttamente comparabile, sono indicati i massimi alfanumerico e numerico. Riferimenti tecnici: ISO/IEC 18004, ISO/IEC 16022, ISO/IEC 16023 e ISO/IEC 15438.

Tutti i codici di cui abbiamo parlato finora mantengono la caratteristica comune di essere composti da componenti a contrasto bianchi e neri, per cui se si volessero associare bits di informazione ad aree di codice in base al fattore cromatico o di

semplice luminosità, ogni valore campionato porterebbe esattamente un solo bit di informazione (bianco/chiaro = 0 e nero/scuro = 1 o viceversa). Questi codici inoltre non sono molto recenti, e per anni hanno avuto poco successo, tuttavia nell'epoca degli smartphones, caratterizzati da ottime fotocamere integrate e da un software di base sempre più vicino ai sistemi operativi tipici dei personal computer, si sta avendo una forte spinta all'utilizzo di dispositivi mobili come lettori ottici, previa installazione di un apposita applicazione. Per questo motivo, come ci si poteva aspettare, il concetto di codice è stato rivisto al fine di sfruttare appieno l'hardware maturato negli ultimi anni: nascono i primi codici bidimensionali a colori, che non hanno nulla a vedere con un discorso estetico, puntando invece ad ottenere maggiore densità di informazione e dunque maggiore capacità di memorizzazione dati su una certa superficie. Appartiene a questa nuova tipologia Microsoft HCCB [1], codice bidimensionale i cui componenti sono righe di triangoli colorati, secondo una tavolozza a 2, 4 o 8 colori, che vanno quindi a rappresentare 1, 2 o 3 bits di informazione per modulo. La tecnologia Microsoft presenta in questo senso delle novità rispetto a codici meno recenti, ma presenta lo svantaggio di essere proprietaria.

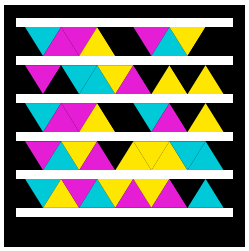

Figura 1.4: Un esempio di codice Microsoft HCCB - Microsoft Tag
Nota editoriale (2026): Questo schema creato dall'autore è stato disegnato indipendentemente per l'edizione digitale; la sequenza cromatica interna differisce dal campione storico. Riferimento tecnico: Microsoft HCCB.

In questo lavoro si presenta il progetto di un codice ad altà capacità a colori che riprende i codici Quick Response, cercando di estendere lo standard che li definisce;

si ritiene infatti importante andare nella direzione dell'innovazione di codici secondo una strada diversa da quella intrapresa da Microsoft, sia perché si vogliono evitare tecnologie proprietarie, sia perché le strutture di codici bidimensionali meno recenti, come quella dei Quick Response, sembrano ottime, con nulla da invidiare a quella del Microsoft HCCB stesso.

L'idea è quindi di rielaborare lo standard dei codici Qr, che possiede delle caratteristiche di grande spessore sia in affidabilità, che in capacità, che in rapidità di decodifica, tale da renderlo il codice predominante in Giappone, dove fu sviluppato, e di garantirgli ampia diffusione nel resto del mondo.

Si cerca nel seguito di questa sezione di dire qualcosa di più sui codici a cui abbiamo fatto finora solo pochi accenni, descrivendone almeno uno per ogni classe, passando poi a parlare dei Quick Response in maniera estesa.

1.2.3 Codici 2d a Barre Impilate: PDF417, Codablock

I codici bidimensionali a barre impilate, sono in sostanza la più diretta evoluzione dei classici codici a barre: possono essere visti come una serie di codici a barre monodimensionali, molto piccoli, impilati uno sopra l'altro (Stacked Symbologies); ne sono esempi i codici PDF417, in cui la sigla 417 indica che l'unità di dati elementare del codice, detta "codeword", è composta da quattro barre e da quattro spazi per un totale di 17 moduli. La capacità del PDF417 è buona, potendo codificare in un simbolo circa 2000 caratteri.

Figura 1.5: Struttura del codice PDF417

Nota editoriale (2026): Questo simbolo è stato generato e annotato indipendentemente dall'autore per l'edizione digitale; non riproduce una figura dello standard. Riferimento tecnico: ISO/IEC 15438.

Il simbolo consiste dalle 3 alle 90 righe, ognuna delle quali è come un piccolo codice a barre; ogni riga inizia con una codeword di controllo che specifica il numero della riga e quale tasso di correzione di errore si sta usando su questa.

Il PDF417 è molto utilizzato nei trasporti, si usa ad esempio per la posta gestita dal servizio postale degli Stati Uniti.

Il Codablock è un altro esempio di codice bidimensionale a barre impilate. Realizzato in Germania, in più versioni, il Codablock è formato da più righe di codici tradizionali come il Code 39 o, nella versione più recente, il Codablock F, come il Code 128. (Ogni riga può avere fino a 64 caratteri di dati più 5 caratteri di controllo). Nonostante sia abbastanza usato in Germania, nel resto del mondo non ha avuto grande diffusione.

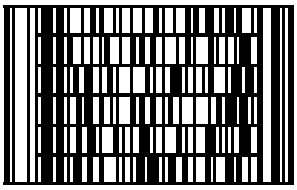

Figura 1.6: Esempio di codice Codablock-F

Nota editoriale (2026): Questo simbolo è stato generato indipendentemente dall'autore per l'edizione digitale e non riproduce il campione raster storico. Riferimento tecnico: AIM Codablock F.

1.2.4 Codici 2d a Matrice

I codici bidimensionali a matrice hanno una struttura abbastanza diversa dai codici a barre, anche se a volte vengono definiti erroneamente come tali. I dati non sono codificati quindi in base a spessori e distanze tra barre, ma in base alla lumonosità di superfici del codice (tipicamente di forma quadrata) che formano le cosiddette celle (o moduli), e che portano usualmente un bit di informazione qualora non svolgano funzioni di controllo. I moduli sono quindi quelle unità elementari alla base della struttura del simbolo stesso. Nonostante siano tecnologie attualmente in uso, non sono senz'altro recenti, in quanto risalgono comunque agli anni 90.

Il codice DataMatrix

Un tipo molto diffuso di codice a matrice è il Data Matrix, un codice bidimensionale composto da moduli bianchi e neri disposti all'interno di uno schema di forma rettangolare o quadrata.

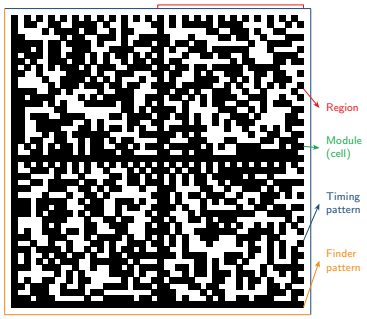

Figura 1.7: Struttura del codice Data Matrix
Nota editoriale (2026): Questo simbolo è stato generato e annotato indipendentemente dall'autore per l'edizione digitale; non riproduce una figura dello standard. Riferimento tecnico: ISO/IEC 16022.

Come si vede in figura 1.7 i moduli del simbolo si trovano all'interno di un perimetro composto dal finder e dal timing pattern: due bordi adiacenti colorati in maniera uniforme costituiscono il primo pattern, gli altri due bordi adiacenti (anche

in questo caso a forma di L) riempiti da un'alternanza di celle bianche e nere formano il secondo.

Il Finder Pattern consente ai dispositivi di lettura ottica di localizzare e orientare correttamente il codice Data Matrix all'interno dell'immagine scansionata: anche se nella scansione del codice, che risente ovviamente dell'errore umano, fossero presenti elementi di disturbo, il lettore ottico deve scartare gli elementi estranei ed essere in grado di riconoscere la sola area della scansione che costituisce il codice, anche qualora questo sia sottosopra o orientato diversamente.

Il Timing Pattern permette di capire il numero di righe e di colonne che costituiscono il simbolo, offrendo quindi una chiave di lettura dell'area interna al codice: si noti a questo proposito, che conoscendo la sola dimensione dell'area codice per come è stata percepita dal sensore ottico, non potremmo dire nulla circa la dimensione di un modulo; grazie all'alternanza di moduli bianchi e neri del timing pattern si può invece superare questo scoglio e procedere col processo di decodifica. Ovviamente se il numero di moduli all'interno del codice fosse fisso non avremmo bisogno del timing pattern per capire quanti moduli compongono il simbolo, tuttavia sarebbe limitante se fosse così: man mano che la dimensione dei dati da codificare aumenta, il numero di righe e di colonne cresce: le dimensioni del codice variano da 8x8 a 144x144 celle.

All'interno dei bordi che costituiscono questi due patterns vi sono quei moduli, organizzati in righe e colonne, che rappresentano i dati codificati, mentre ovviamente le celle costituenti finder e timing patterns, svolgendo la loro funzione di controllo, non portano alcun bit di informazione.

La capacità dei simboli Datamatrix è molto buona, riuscendo a codificare dati per dimensioni che vanno da pochi bytes a circa 2 kilobytes, ma è ragionevole chiedersi come si fa a garantire una certa affidabilità per quantità di dati così cospicue: tutti

i simboli che utilizzano il sistema di correzione degli errori ECC200 si avvalgono di un sofisticato meccanismo, basato sull'algoritmo di Reed-Solomon, che permette la ricostruzione di una percentuale di dati erronei che può spingersi fino al 30%, cosa impossibile per i classici codici a barre lineari. Ovviamente tutto ciò ha un costo: tanto più si desidera potenziare la capacità di correggere gli errori, tanto maggiore sarà la quantità necessaria di dati ridondanti da codificare nel simbolo, che dunque sottrarrà superficie del codice altrimenti spendibile per codificare dati addizionali.

Il codice Quick Response (QR)

Per quanto si è detto in riferimento ai codici DataMatrix, possiamo senz'altro affermare che tali codici hanno un'ottima struttura, con caratteristiche di qualità; tuttavia in questo lavoro si è scelto di riprendere ed estendere un altro tipo di codice, il Quick Response. L'atto di scegliere quale codice rielaborare e riprogettare per generare un codice ad alta capacità che sfruttasse l'informazione cromatica non poteva non prendere in considerazione le qualità del codice: il massimo risultato si sarebbe avuto lavorando sul codice che più di altri avesse avuto caratteristiche di elevata capacità, affidabilità e velocità di decodifica (oltre a una buona diffusione e dunque popolarità). Nonostante il DataMatrix fosse un ottimo candidato, anche il codice Quick Response possiede finder e timing patterns (seppur diversi in forma da quelli del DataMatrix, con analoghe funzioni), come anche un sistema sofisticato di correzione degli errori, elevata affidabilità, rapidità di lettura e ampia popolarità nel mondo. Nel seguito della trattazione si parlerà in maniera estesa dei codici Quick Response, mentre si è inserito solo un accenno in questa sezione per ricordare da un lato che si tratta di un codice bidimensionale a matrice, dall'altro che nonostante sia stato preso come riferimento per il lavoro, scelto tra tutti i codici esistenti, anche il DataMatrix appena

discusso sarebbe potuto essere un buon candidato.

1.2.5 Codici 2d a colori: Microsoft HCCB

Si è detto più volte nel corso della trattazione, che le tecnologie trattate sono tutt'altro che recenti, tuttavia cercare di portare innovazione in un ambito per il quale la ricerca è stata sopita per lungo tempo non è un'idea così strana se si pensa che la stessa Microsoft in questi anni sta promuovendo un codice proprietario chiamato High Capacity Color Barcode (HCCB) [1] [2], che come dice il nome stesso utilizza l'informazione cromatica per ottenere un'elevata capacità di memorizzazione dati. Codici di questo tipo non devono essere intesi come competitors odierni di codici tradizionali già diffusi, altamente affidabili per l'uso che se ne fa nel campo dei trasporti, della gestione dei magazzini, etc... tuttavia costituiscono un buon metodo per dare una rappresentazione ottica di grandi quantità di dati in poco spazio, ancora affidabile grazie ai progressi dell'hardware dei sensori ottici moderni. Si ricorda infatti che al giorno d'oggi ogni smartphone è potenzialmente un lettore ottico di buona qualità, considerando le ottime fotocamere a disposizione su tali dispositivi.

Il codice HCCB sceglie di usare gruppi di triangoli colorati anziché le celle quadrate di cui abbiamo discusso parlando di codici bidimensionali tradizionali, come il Data Matrix. Nonostante sia possibile generare codici in bianco e nero, si può tentare di incrementare la densità di informazione utilizzando schemi a 4 o 8 colori, che rispettivamente raddoppiano o triplicano la quantità di dati codificabile in una certa superficie rispetto al caso base del bianco e nero (in generale di qualunque schema a 2 valori).

Microsoft afferma di essere riuscita a codificare 3500 caratteri alfanumerici per pollice quadrato, utilizzando stampanti e sensori ottici abbastanza comuni.

Al di là di quello che sono le reali prestazioni, ammesso siano buone quanto le dichiarazioni in merito, Microsoft ha promosso ultimamente una implementazione di HCCB a 4 colori strutturata in una griglia di 5 x 10 triangoli chiamata Microsoft Tag, dunque un caso particolare di codice HCCB. Potendo portare solo 13 bytes di dati, inclusa la ridondanza necessaria per l'algoritmo di correzione degli errori di Reed-Solomon, tale codice è essenzialmente un web link machine-readable: quando il lettore ottico decodifica il codice, l'applicazione invia sulla rete le informazioni lette ai servers Microsoft, che risponde con l'url associato; dopodiché il browser dell'utente può dirigersi sul web site di interesse.

Dunque possiamo interpretare il meccanismo dicendo che il codice Microsoft Tag contiene in realtà solo l'identificativo di un contenuto che de facto va recuperato necessariamente tramite la rete Internet. Dunque, a cosa è servito utilizzare per uno schema di questo tipo un codice nato dal progetto di incrementare la densità di informazione tramite i colori? Nella pratica, a meno di motivazioni commerciali, un sistema del genere si sarebbe potuto realizzare mediante codici già esistenti da molti anni, poiché nonostante sia vero che si ottiene un codice stampabile di dimensione inferiore rispetto a codici in bianco e nero, il vantaggio risulta minimizzato nella codifica di soli 13 bytes. In figura 1.4 avevamo in realtà visto un'istanza di HCCB che costituiva proprio un Microsoft Tag: possiamo notare la struttura fortemente orizzontale del codice caratterizzato da 5 righe di 10 triangoli l'una, divise le une dalle altre da apposite linee bianche (svolgono funzioni di controllo come ad esempio rilevare l'orientazione del codice, non portano informazioni).

Versioni di codici a 4 ed a 8 colori diverse dai Microsoft Tags possono codificare ovviamente un maggior quantitativo di informazioni, come si vede in figura 1.8.

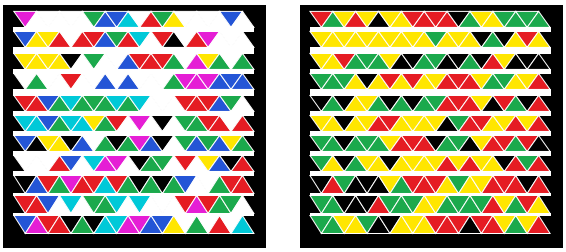

Figura 1.8: Codici Microsoft HCCB a 4 ed a 8 colori

Nota editoriale (2026): La geometria e la tavolozza seguono l'oggetto discusso nella tesi del 2010, ma per questa edizione l'autore ha cambiato il colore di dodici singoli triangoli in ciascun pannello. Riferimento tecnico: Microsoft HCCB.

Capitolo 2

Il Codice Quick Response (ISO/IEC 18004)

Nota editoriale (2026)

Per prudenza editoriale, questa edizione omette il Capitolo 2 della tesi del 2010, che descrive in dettaglio elementi della specifica QR Code.

La simbologia a colori discussa nella tesi del 2010—denominata *High Capacity Colored Two-Dimensional Codes* (HCC2D) soltanto nelle pubblicazioni successive alla tesi—riutilizza la struttura a matrice quadrata di QR Code: finder pattern, alignment pattern, timing pattern, format information, version information, formule di mascheramento e comportamento della correzione d'errore Reed–Solomon compatibili con QR Code / ISO/IEC 18004:2006, salvo le diverse regole della simbologia a colori descritte nei capitoli successivi.

I capitoli successivi richiamano tali strutture solo dove necessario e descrivono il lavoro specifico della simbologia a colori, denominata HCC2D nelle pubblicazioni successive.

"QR Code" è un marchio registrato di DENSO WAVE INCORPORATED in Giappone e in altri Paesi. Né questo lavoro accademico né le successive pubblicazioni su HCC2D sono sponsorizzati, approvati o affiliati a DENSO WAVE INCORPORATED. Si tratta di una scelta prudenziale relativa alla ripubblicazione e non modifica il rapporto tecnico sopra dichiarato.

Capitolo 3

Quick Response a Colori ad Alta Capacità

Nota editoriale (2026): La tesi del 2010 utilizzava diverse espressioni descrittive per la simbologia proposta, tra cui "Quick Response ad alta capacità," "Quick Response a colori ad alta capacità," "QR/Quick Response a colori," le relative varianti con "maggiore capacità" o "alta densità," e i corrispondenti riferimenti alla nuova simbologia a colori, ai simboli, al lettore, al generatore e ai moduli software. La denominazione "HCC2D" fu adottata nelle pubblicazioni successive alla discussione. In tutta questa edizione, tutte queste espressioni storiche si riferiscono al formato successivamente denominato HCC2D. I riferimenti a QR/Quick Response hanno carattere esclusivamente descrittivo: "QR Code" è un marchio registrato di DENSO WAVE INCORPORATED in Giappone e in altri Paesi, e il loro uso non implica sponsorizzazione, approvazione o affiliazione, né che la simbologia proposta sia essa stessa un QR Code.

Questo è un capitolo centrale del lavoro di tesi, volto a ideare e progettare un'evoluzione dello standard dei codici Quick Response ("QR"), che abbia il fine di incrementare la densità di informazione dell'area di codifica del codice, tentando di avvalersi dell'informazione cromatica, anziché fermarsi a valutare la sola luminosità come avviene per i codici QR standard. Si vuole quindi proporre un codice bidimensionale a colori ad alta capacità [4], che trae origine dalla struttura dei codici Quick Response.

I codici Quick Response standard hanno ottime caratteristiche di rapidità di lettura, affidabilità e capacità, come si è potuto capire durante la lettura dei capitoli precedenti. La nuova spinta all'utilizzo delle tecnologie di codici a barre e non, nasce dal fatto di non dover più necessitare di hardware ad-hoc per loro lettura, essendo ormai disponibili sul mercato tanto ottime quanto economiche fotocamere sulla maggior parte dei dispositivi mobili. La tendenza non mostra segni di rallentamento, nel senso che di anno in anno si producono fotocamere sempre migliori a costi contenuti. In questo contesto, il contributo del lavoro vuole essere indirizzato ad avvalersi fino

in fondo di hardware così performanti, riprogettando dei codici che, seppure tuttora attuali, non furono pensati per gli scenari dei nostri giorni quando furono ideati, progettati e realizzati. Sarebbe stato infatti impossibile prevedere il proliferarsi di dispositivi mobili, come gli odierni smartphones, in un periodo, come quello degli anni '90, in cui a malapena si vendevano i primi costosi telefoni cellulari. Negli ultimi anni Microsoft ha introdotto un codice a colori, High Capacity Color Barcode (HCCB), (di cui si è discusso in sezione 1.2.5), che rappresenta un primo tentativo di sfruttare l'informazione cromatica nei codici per incrementarne la densità di informazione, tuttavia rimane una soluzione proprietaria che non sembra dimostrare punti di forza tali da mettere in ombra codici bidimensionali ben strutturati come Datamatrix o Quick Response: in questo capitolo si cercherà di spiegare come rielaborare ed adattare il codice Quick Response al fine di andare incontro all'esigenza di sfruttare anche l'informazione cromatica per una maggiore capacità del simbolo, mantenendo allo stesso tempo una sufficiente affidabilità ereditata dall'ottima struttura dei codici Quick Response standard.

3.1 Obiettivi del Progetto

Cercare di definire una nuova simbologia di codici bidimensionali basata sui Quick Response, dicendo che tramite il colore si vogliono ottenere codici ad elevata densità di informazione, non completa il quadro degli obiettivi che ci si propone di raggiungere con il progetto. Fermo restante questo punto come obiettivo primario, il progetto ha lo scopo di generare la nuova simbologia puntando ad una struttura quanto più vicina possibile ai codici Quick Response standard, tale che si possano considerare i nuovi simboli un'evoluzione diretta dei codici standard, non dei competitors. A questo proposito si vuole raggiungere un obiettivo più stringente, ovvero progettare i

nuovi codici in maniera tale che, essendo questi un'estensione, un'evoluzione diretta dei codici qr standard, si abbia che l'insieme dei codici qr standard sia un sottoinsieme dei codici appartenenti alla nuova simbologia, oppure detto in altre parole, si possa dire che un codice qr standard sia un caso particolare di codice qr a colori ad alta densità, nel caso in cui si usino solo due colori. L'obiettivo di generare l'insieme dei nuovi codici come superset rispetto all'insieme dei codici qr classici consente di ottenere uno scopo molto importante: mantenere retro-compatibilità con lo standard, facendo in modo quindi che un lettore di codici QR a colori ad alta capacità continui a saper decodificare i codici tradizionali. Come ultimo punto, ma non meno importante, la nuova simbologia, se da un lato incrementa la densità dei dati, non deve smorzare eccessivamente le caratteristiche di affidabilità o rapidità di decodifica tipiche dei codici QR, né tantomeno deve richiedere hardware specifici o poco comuni o troppo costosi: il fallimento nel raggiungere gli obiettivi suddetti renderebbe de facto inutilizzabile la nuova simbologia di codici.

3.1.1 Project Scope

I risultati tangibili che ci si aspetta dal lavoro sono sia la definizione delle specifiche riguardanti la struttura della nuova simbologia di codici, nonché la realizzazione del generatore e del lettore ottico corrispondente. Tali risultati devono essere applicabili sia in ambiente desktop che mobile, poiché come si è detto nel corso della trattazione, i dispositivi mobili ricoprono un ruolo centrale nella spinta all'evoluzione dei codici bidimensionali che si è avuta negli ultimi anni, pertanto è fondamentale che i nuovi codici siano utilizzabili in tale contesto.

3.2 Problematiche Aperte

Numerose problematiche vengono alla luce nel momento in cui si cerca di evolvere la simbologia Quick Response; è necessario ragionare attentamente prima di proporre ogni eventuale introduzione di nuovi elementi o modifica di quelli esistenti, in quanto si rischia di aggiungere nuove funzionalità a spese di altre preesistenti e performanti, invalidando nel caso peggiore i processi critici alla base della decodifica dei simboli, fatto che renderebbe vano il lavoro di evoluzione dei codici: simboli ad elevata capacità, tuttavia quasi illeggibili, sarebbero sicuramente da considerarsi un passo indietro. Il primo punto su cui riflettere riguarda capire cosa rielaborare per introdurre il colore nei Quick Response e cosa invece può essere mantenuto esattamente per come definito nello standard. Potrà capitare inoltre che apporre modifiche ad elementi preesistenti abbia come side effect di rendere non più ben definito qualche concetto correlato: di conseguenza possono nascere sfide teoriche importanti per il minimo cambiamento, come spesso avviene quando si manipola un prodotto complesso, come è nel nostro caso la simbologia dello standard Quick Response. In questo paragrafo si faranno ragionamenti sui diversi tasselli che compongono la struttura dei codici QR, cercando di lanciare possibili idee per definire le nuove specifiche, con l'intento mirato di mettere in luce ogni possibile complicazione che possa sorgere a causa di eventuali modifiche allo standard, mentre le scelte intraprese saranno spiegate nei paragrafi successivi.

3.2.1 Osservazioni sui Function Patterns

In prima analisi ragioniamo sui functions patterns; in sezione 2.2 sono stati ampiamente discussi position, timing e alignment patterns, elementi strutturali costitutivi dei codici QR standard. L'introduzione del colore in che maniera impatta su tali elementi? Questi rimangono ancora validi o le funzionalità offerte vengono invalidate?

Sono nel complesso sufficienti o è necessaria l'introduzione di qualcosa di nuovo? Ci si può facilmente convincere come il voler sfruttare l'informazione cromatica nei codici per aumentare la densità di informazione sia un problema ortogonale a quanto svolto dagli algoritmi di image processing nel tentativo di leggere il simbolo nella scansione, spesso riuscendo a venirne accapo anche quando vi siano pronunciate distorsioni. Quindi, il supporto offerto dai function patterns dello standard a tali algoritmi non dovrebbe essere impattato in uno scenario di introduzione dei colori, fatto che ci porta a considerare di mantenere questi patterns così come sono per almeno due ragioni: in primo luogo mantenendoli si va verso il raggiungimento dell'obiettivo di retro-compatibilità, in secondo luogo la loro struttura è frutto di grande esperienza e garantisce ottime prestazioni in affidabilità. In fondo è anche per queste loro caratteristiche che si è scelto di lavorare ad una nuova simbologia a colori basata sui Quick Response piuttosto che fondata su altri tipi di codici, dunque perché non preservarle?!. Fermo restando il concetto di ereditare i patterns esistenti, le nostre nuove esigenze potrebbero richiedere l'introduzione di nuovi elementi di controllo; sarebbe forse utile un pattern che possa consentire di contrastare le distorsioni cromatiche della scansione percepita dal lettore ottico, fenomeno in relazione al quale codici che usino schemi a 4 o più colori sono ben più sensibili di quanto non avvenga in simboli in bianco e nero.

3.2.2 Osservazioni sull'Area di Codifica

Per quanto riguarda l'area di codifica, si è visto come questa sia divisa in area dati, informazioni di formattazione e di versione, e per ognuna delle tre regioni si è scesi in dettaglio in sezione 2.3, nei rispettivi paragrafi. Considerando che è sicuramente in questa area che si concentreranno le novità introdotte dall'aggiunta dei colori per l'alta capacità, quali riflessioni possiamo fare sull'evoluzione della regione di codifica?

Non avrebbe probabilmente molto senso modificare la struttura delle informazioni di formattazione e di versione(qualora presente), prima di tutto perché ci allontana dal centrare l'obiettivo di retrocompatibilità per cui sarebbe difficile dire che i codici QR standard siano sottoinsieme stretto della nuova simbologia di codici, tuttavia non è solo questo punto che fa sorgere dubbi circa la necessità di modifica di tali aree, infatti avrebbe ad esempio poco senso cercare di restringerne la superficie allocata aumentando la densità dei dati mediante le sfumature cromatiche quando invece ha molto più senso privilegiare l'affidabilità. Se si considera che tali aree codificano pochi bits e la loro corretta lettura è così critica da rischiare di invalidare l'intero processo di decodifica, ci si accorge che un minimo vantaggio non vale un così grande rischio, dunque perché non ereditare le specifiche di tali aree direttamente dallo standard, rappresentati da elementi a contrasto nelle sole gradazioni di bianco e nero?!

Il terzo campo che trattiamo, indicato in figura 2.1 con la dicitura "Data and Error Correction Codewords", [1] e trattato in dettaglio nel paragrafo 2.3.3, è la superficie allocata alla codifica dei dati che abbiamo interesse a rappresentare; in considerazione degli alti tassi di errore che sempre si hanno quando si scende ad un livello così basso da trattare percezioni su uno strato fisico ad opera di un sensore hardware (nel nostro caso il lettore ottico), i dati da codificare non sono i soli desiderati: a questi infatti va aggiunta una certa quantità di ridondanza atta a gestire gli errori, a cui si fa riferimento proprio con "Error Correction Codewords", ovvero codewords per la correzione d'errore.

L'intera superficie preposta alla codifica dei dati ridondati sarà visibilmente diversa da quelle relative ai codici standard, poiché è qui che avremo elementi a contrasto di varie sfumature cromatiche anziché celle bianche e nere, rappresentanti non più un

[1] Si ricorda che nel contesto dello standard Quick Response per codeword si intende de facto un byte, dunque 8 bits.

solo bit per modulo come avviene nei simboli classici, bensì 2,3 o 4 bits per modulo rispettivamente secondo schemi a 4,8 o 16 colori.

3.2.3 Principali Problematiche e Sfide da Affrontare

Poiché ci dirigiamo in una direzione che diverge dallo standard, si aprono diversi interrogativi su come raffinare le idee al fine di definire delle specifiche per la nuova simbologia che non presentino incongruenze. Vogliamo esaminare in questa sezione alcune sfide da affrontare per il raggiungimento degli obiettivi relativi alla definizione della nuova simbologia di codici; scopo del paragrafo non sarà di descrivere in maniera esaustiva tutti i problemi da risolvere presentandoli in lunghi elenchi, bensì focalizzare l'attenzione sui temi principali, facendo dei ragionamenti e lanciando degli interrogativi. Poniamoci quindi i dubbi che giustamente devono sorgere quando si tenta di apportare modifiche ad un prodotto abbastanza complesso come può essere il codice Quick Response, mentre nelle sezioni successive cercheremo di presentare possibili soluzioni agli interrogativi sollevati.

1. Esistono delle tabelle relative ai codici Quick Response standard che mettono in relazione grandezze come le codewords totali, la versione del simbolo, il livello di correzione d'errore, il tipo di blocchi di Reed-Solomon in cui è suddivisa l'area dati, etc ... queste non tengono ovviamente in considerazione il parametro relativo ai bits rappresentati da ogni modulo, poiché per lo standard ogni cella rappresenta sempre 1 bit; la nuova simbologia proposta è invece sensibile a tale grandezza. Nel contesto dei nuovi codici, un simbolo aumenta la sua capacità di memorizzare informazioni quanto più cresce la cardinalità dello schema di

colori usato in codifica; di conseguenza si dovrebbero aggiornare tali tabelle per tenere da conto l'impatto di questo nuovo parametro.

2. Se è vero che vogliamo mantenere retro-compatibilità coi codici Quick Response standard, e che quindi non vogliamo modificare la struttura delle regioni e dei patterns preesistenti, abbiamo però il problema di non sapere come codificare l'informazione di quanti colori stia utilizzando un codice appartenente alla nuova simbologia (2, 4, 8... colori!?), informazione di cui però il lettore ha estrema necessità al fine di poter interpretare correttamente i moduli; andrebbe quindi introdotto un campo apposito?

3. Ammesso che il lettore ottico possa ottenere la cardinalità dello schema di colori utilizzati dal generatore, questi avrebbe ancora il problema di non sapere quali colori siano stati effettivamente usati in fase di codifica; lo schema di colori adoperato in generazione andrebbe preso come riferimento in decodifica al fine di poter elaborare correttamente i moduli colorati dell'area dati. Questi colori dovrebbero essere noti a priori al lettore? Come possiamo gestire la distorsione cromatica causata dalla fotocamera?

4. Il mascheramento dei dati, utile ad incrementare l'affidabilità dei codici, diventa un procedimento non più del tutto ben definito se ci poniamo in un contesto in cui sono introdotti i colori. Si ricorda a tale proposito quanto sia importante evitare la generazione di simboli scarsamente leggibili, fatto decisamente negativo che può capitare a fronte di alcuni inputs specifici particolarmente "sfortunati"; ci sono diverse ragioni perché ciò possa avvenire, ne ricordiamo solo una a titolo di esempio (si veda il capitolo 2 per una più ampia trattazione); consideriamo degli inputs che diano luogo a codici che non posseggano un ragionevole

bilanciamento di proporzioni tra celle nere e bianche: un simbolo scuro per il 90% della sua superficie diviene molto meno affidabile in lettura di quanto non accada con un codice ben bilanciato (50% e 50%) relativamente alle aree chiare e scure). Per approfondimenti relativi alla funzionalità di masking ci si rivolga al capitolo 2, ad ogni modo cercando di sintetizzare in maniera semplice, seppur non esaustiva, si può dire che lo standard stabilisce quanto segue:

- Si generino dei mask patterns di tipo matriciale e di dimensioni pari a quelle del simbolo da generare quanto a numero di celle che lo compongono. Il mask pattern sarà quindi una matrice di celle in cui un modulo è scuro se una certa condizione è vera [ad esempio se (indice di riga + indice di colonna)mod2=0, oppure se (indice di colonna)mod3=0, etc...], chiaro altrimenti. Si hanno 8 maschere alternative da produrre, ognuna associata ad una condizione specifica che ne costituisce il principio di generazione.
- Il codificatore valuti la bontà di ciascuna delle 8 maschere guardando al risultato dello xor tra ognuno degli 8 mask patterns generati ed il simbolo non mascherato, al fine di scegliere poi il migliore per il caso specifico. Cosa si intende per risultato migliore? Per capire quale sia la maschera più adatta per lo specifico codice Quick Response ci sono dei criteri ben definiti dallo standard che danno un punteggio ad ogni maschera.

Quindi, se è vero che in fase di decodifica a moduli di una data luminosità si associano valori nel dominio {dark, light} in primis, mappati successivamente coi rispettivi valori del dominio {0, 1}, a seguito di una conversione analogico digitale (si noti a tale proposito che le soglie di luminosità tramite le quali si definisce quali celle vadano interpretate come chiare o scure sono adattative) si

può poi fare lo xor con i valori ottenuti nei mask patterns, generati dalle diverse condizioni, e di nuovo stabilire in base al risultato di questo se un modulo debba essere chiaro o scuro.

Tornando all'idea dei codici Quick Response ad alta capacità proposti, se ad un modulo è associato più di un bit come procedere col mascheramento? Se normalmente, un modulo nero viene invertito con uno bianco e viceversa, in uno schema a più colori cosa sarebbe opportuno che accada?

5. Dovendo riconoscere le diverse sfumature cromatiche, come calcoliamo le distanze tra i vari colori? Avremo sufficiente precisione nel riconoscere colori "simili" in un'immagine di bassa qualità, tanto da riuscire a decodificare il simbolo? Ad ogni modo appare ovvio che, come accennato, occorrerà definire una metrica di similarità tra i colori per poter rendere concreto il concetto di "similarità" tra di essi.

3.3 Specifiche di Progetto Proposte

Sono state trovate varie soluzioni alle problematiche sollevate circa una possibile evoluzione dei codici QR che porti alla definizione di codici Quick Response a colori ad alta capacità. Queste soluzioni non sono di sicuro le uniche possibili, ma sono quelle che cercano di centrare gli obiettivi descritti in sezione 3.1; nel complesso si cercherà di produrre delle specifiche per la nuova simbologia che abbiano le seguenti caratteristiche:

- Siano il più possibile "lineari" con lo standard costituendone una diretta evoluzione. Dovrebbero cioè ereditare ogni peculiarità dello standard che non sia strettamente necessario rielaborare in relazione alle nuove esigenze.

- Perseguano un obiettivo di semplicità, che consenta di convincersi ragionevolmente di non introdurre incongruenze nell'elaborazione di un prodotto complesso come il Quick Response standard, che allo stato dell'arte presenta delle buone prestazioni che non si vuole rischiare di compromettere nel tentativo di evolverlo.

3.3.1 Estensione delle Tabelle dei Codici Quick Response

In risposta alla problematica contrassegnata con (1.) in sezione 3.2.3, è necessario tenere in considerazione il nuovo parametro relativo a quanti bits codifichi ciascun modulo del simbolo; dobbiamo ragionare su come aggiornare nella maniera più diretta possibile le tabelle relative ai codici Quick Response definite nello standard ISO/IEC 18004, le quali mettono in relazione grandezze come le codewords totali, la versione del simbolo, il livello di correzione d'errore, il tipo di blocchi di Reed-Solomon in cui è suddivisa l'area dati, etc ... ma non i bits rappresentati da ogni cella, essendo sempre valido nei codici Quick Response classici la regola secondo cui ogni modulo rappresenta un solo bit di informazione.

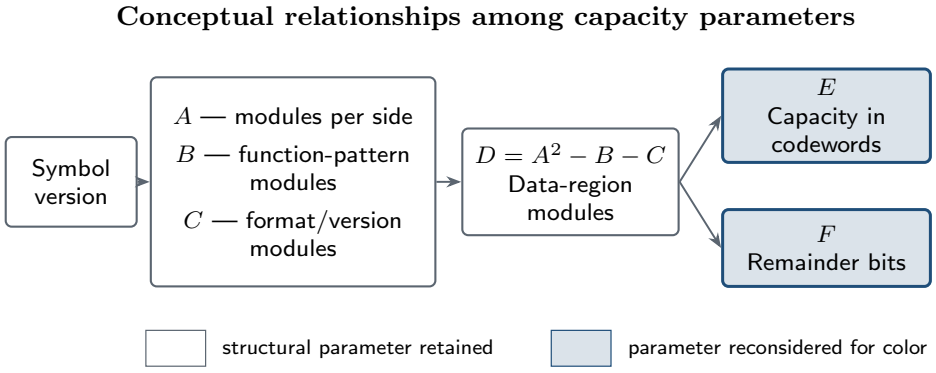

Figura 3.1: Relazioni concettuali fra i parametri di capacità discussi nel testo del 2010.

Nota editoriale (2026): La tabella numerica della figura del 2010 non è riprodotta in questa edizione digitale pubblica. Per rendere comprensibili i richiami successivi, le sue colonne erano, da sinistra: (1) Version; (2) No. of Modules/side (A); (3) Function pattern modules (B); (4) Format and Version Information modules (C); (5) Data modules except (C), $D = A^2 - B - C$; (6) Data capacity [codewords] (E); (7) Remainder Bits. Nel nuovo schema le colonne 1–5, mantenute, sono rappresentate in bianco; le colonne 6–7, proposte per la parametrizzazione, sono rappresentate in celeste. Il nuovo schema mostra soltanto le relazioni concettuali, senza valori della tabella. Riferimento tecnico: ISO/IEC 18004:2000.

In figura 3.1 è riportato un estratto relativo alla tabella 1 del documento relativo allo standard Quick Response di riferimento, la quale illustra per ogni versione:

1. Il numero di moduli che compongono la matrice costitutiva del simbolo su una dimensione. Esempio: alla versione 1 corrispondono 21 moduli, dunque il simbolo sarà esattamente di 21x21 (441) moduli totali.

2. Il numero di moduli del simbolo che vengono impiegati per i function patterns, dunque che non portano bits di informazione.

3. Il numero di moduli del simbolo impiegati per rappresentare le informazioni di versione e formattazione.

4. Il numero di moduli della regione di codifica impiegati effettivamente per la rappresentazione dei dati di interesse. Questi, come si è detto più volte nel corso della trattazione, sono comprensivi della ridondanza necessaria in grado di supportare la funzionalità di correzione d'errore.

5. Il numero di codewords, ognuna equivalente a un byte, rappresentabili nel simbolo. Poiché anche in questo caso è inclusa la ridondanza, la quantità massima di dati per l'utente finale che si possono codificare nel simbolo sarà in alcuni casi anche molto minore di tale valore.

6. Il numero di bits di resto che saranno de facto inutilizzabili. Poiché la quantità di informazione portata, espressa in bits, può non essere un valore multiplo di 8 può capitare di avere dei bits residui superflui.

I campi relativi alla tabella di cui abbiamo discusso, dovrebbero quindi essere aggiornati per tenere in considerazione quel parametro aggiuntivo dato dal numero di bits rappresentati da ciascun modulo; si vuole quindi riflettere su quali campi estendere e quali mantenere (le colonne per cui verrà proposta la parametrizzazione sono state evidenziate con una colorazione differente in figura 3.1).

I primi attributi della tabella sembrano ancora validi, si possono mantenere il numero di moduli nel simbolo, così come i valori relativi ai function patterns ed alle aree di informazione di versione e formattazione; questo perché come si è detto si sta cercando di ereditare tutto ciò che non è impattato dall'introduzione degli schemi cromatici della nuova simbologia.

La capacità, espressa in colonna E, dovrebbe invece aumentare in base al numero di colori presenti nello schema, in quanto questa caratteristica rappresenta proprio il fondamento dell'idea di evoluzione dei codici: molto semplicemente si potrebbe parametrizzare la colonna E aggiungendo il fattore moltiplicativo dato dai bits rappresentati da ciascun modulo.

Per quanto riguarda il numero di Remainder Bits (espressi nell'ultima colonna), si potrebbe moltiplicare anche questa colonna per il numero di bits per modulo, così

come si è pensato poc' anzi per il valore relativo alla capacità. A questo proposito va aggiunta però un'osservazione, e cioè, qualora il prodotto tra i Remainder Bits e i bits per modulo equivalga o superi 8, si potrebbero aggiungere i bytes in tal modo ricavati al valore di Data Capacity precedentemente calcolato.

$$RemainderBits_{new} = RemainderBits_{old} * BitsPerModule$$
$$DataCapacity_{new} = DataCapacity_{old} * BitsPerModule$$

oppure (minimizzando gli sprechi):

$$RemainderBits_{new} = (RemainderBits_{old} * BitsPerModule) mod 8$$
$$DataCapacity_{new} = (DataCapacity_{old} * BitsPerModule) + (RemainderBits_{old} * (BitsPerModule)/8)$$

Al solo scopo di semplificare la parametrizzazione dei valori relativi alle tabelle, considerato anche che la seconda soluzione pur essendo migliore consente un beneficio minimo, si proseguirà nella trattazione considerando valida la prima assunzione.

Numero di Caratteri del Simbolo e Capacità per i Dati di Input

Trattiamo ora, in figura 3.2, una tabella che relaziona ogni versione del codice con le corrispondenti data capacities relative a ciascuna delle diverse modalità (i cosiddetti "Modes"), ovvero la numerica, l' alfanumerica, quella che lavora con rappresentazioni di caratteri ad 8 bits, ed infine quella dei caratteri Kanji; per ulteriori approfondimenti

sui “Modes” dello standard Quick Response o per informazioni correlate si veda il capitolo precedente.

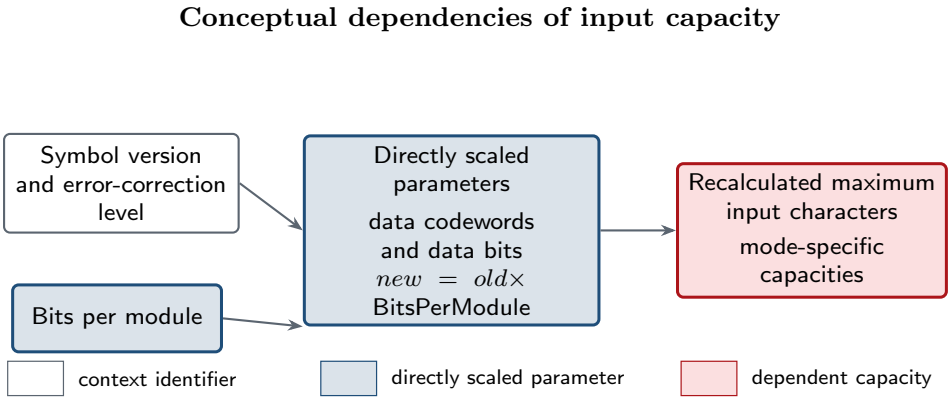

Figura 3.2: Dipendenze concettuali fra versione, correzione d’errore, quantità di dati e capacità in ingresso.

Nota editoriale (2026): La tabella numerica della figura del 2010 non è riprodotta in questa edizione digitale pubblica. Le sue colonne erano, da sinistra: (1) Version; (2) Error correction level; (3) Number of data codewords; (4) Number of data bits; quindi, sotto Data capacity, (5) Numeric; (6) Alphanumeric; (7) 8-bit Byte; (8) Kanji. Le colonne 1–2 erano bianche e fungevano da identificatori. Le colonne 3–4, rappresentate in celeste nel nuovo schema, erano da modificare direttamente. Le colonne 5–8, rappresentate in rosa, erano da ricalcolare in conseguenza delle prime. Il nuovo schema conserva soltanto questa dipendenza concettuale, senza valori della tabella. Riferimento tecnico: ISO/IEC 18004:2000.

Per consentire lo sviluppo di codici Quick Response a colori a maggiore capacità anche questa tabella va parametrizzata (si noti come in generale si stia cercando di aggiornare le tabelle piuttosto che ricostruirle da zero al fine andare nella direzione di una evoluzione diretta che sia retro-compatibile coi codici standard). Le colonne di figura 3.2 che anziché essere a sfondo bianco posseggono una colorazione sono state identificate come aree che necessitano di modifica; si noti come mentre in figura 3.1 alcune colonne venivano ereditate direttamente, per quanto riguarda la tabella di figura 3.2 ogni campo va in qualche modo aggiornato. Sono state inoltre fatte due evidenziazioni in colore differente per sottolineare il fatto che in realtà le colonne della regione di destra (a sfondo rosa) sono da estendersi piucchealtro perché correlate alle colonne della regione di sinistra (a sfondo blu), che andremo a modificare direttamente. Proprio a causa di questo, indirettamente si dovranno andare ad aggiornare anche le

colonne relative alla regione di destra della figura, ovvero le data capacities relative ai vari "Modes".

La maniera più semplice di tenere in considerazione la grandezza data dai bits rappresentati da ogni modulo è anche in questo caso una parametrizzazione semplice come quella vista in precedenza, data da un semplice fattore moltiplicativo:

$$NumberOfDataCodewords_{new} = NumberOfDataCodewords_{old} * BitsPerModule$$

$$NumberOfDataBits_{new} = NumberOfDataBits_{old} * BitsPerModule$$

Per quanto riguarda le colonne di Data Capacity relative ai vari modes (numeric, alphanumeric, 8 bit-byte, kanji), dunque quelle costitutive dell'area che è stata contrassegnata con lo sfondo rosa in figura 3.2, queste vanno ricalcolate in base al valore aggiornato di Number of Data Codewords che è stato poc'anzi definito.

Esempio. In alphanumeric mode si fa in modo che 2 caratteri vengano memorizzati in 11 bits, per cui la Data Capacity relativa è approssimata dal valore ((Number of Data Codewords * 8)/ 11 * 2). In tal senso, una volta che si è modificato il campo relativo alle data codewords ed ai data bits, vanno aggiornate anche le data capacity in accordo ai vari modes; le modalità in cui effettuare tali calcoli per i nuovi valori saranno dunque le stesse normalmente utilizzate dallo standard.

Caratteristiche relative alla Correzione d'Errore

Trattiamo ora un'ulteriore tabella presa dal documento relativo allo standard ISO/IEC 18004, questa volta relativa alla funzionalità di correzione d'errore, che

si ribadisce essere assolutamente fondamentale per avere garanzie di affidabilità nell'utilizzo di codici bidimensionali.

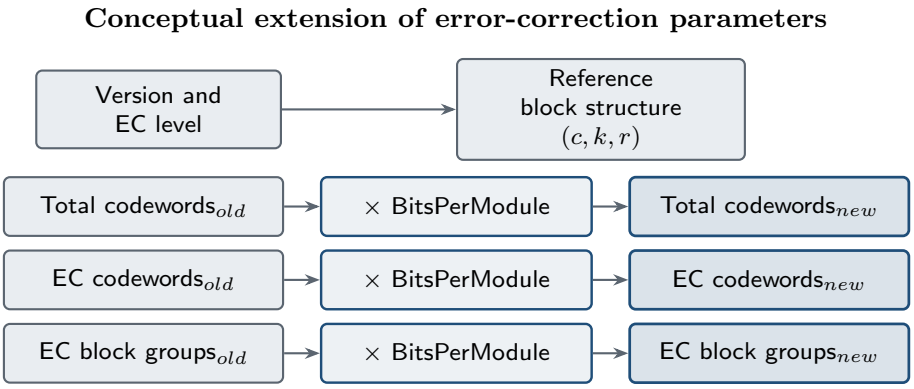

Figura 3.3: Relazioni concettuali impiegate nell'estensione dei parametri di correzione d'errore.

Nota editoriale (2026): La tabella numerica della figura del 2010 non è riprodotta in questa edizione digitale pubblica. Le sue colonne erano, da sinistra: (1) Version; (2) Total number of codewords; (3) Error correction level; (4) Number of error correction codewords; (5) Number of error correction blocks; (6) Error correction code per block. Le colonne 2, 4 e 5 avevano sfondo grigio ed erano quelle da parametrizzare; le colonne 1, 3 e 6 avevano sfondo bianco ed erano considerate ereditabili. Il nuovo schema mostra soltanto il flusso concettuale; i valori dell'esempio restano nel testo. Riferimento tecnico: ISO/IEC 18004:2000.

In maniera analoga alle precedenti la parametrizzazione proposta si riduce ad un semplice fattore moltiplicativo, dunque con poco sforzo ridefiniamo nel seguente modo quelle grandezze i cui valori sono stati valutati inadeguati per accogliere l'evoluzione dei Quick Response proposta:

$$TotNumOfCodewords_{new} = TotNumOfCodewords_{old} * BitsPerModule$$

$$NumOfErrorCorrCwords_{new} = NumOfErrorCorrCwords_{old} * BitsPerModule$$

$$NumOfErrorCorrBlocks_{new} = NumOfErrorCorrBlocks_{old} * BitsPerModule$$

Finora abbiamo soltanto detto come sarebbe possibile aggiornare tali tabelle, tuttavia non sono stati dati riferimenti concreti capaci di convincere di quanto espresso in quest'ultima sezione; pur non essendo intenzionati a fornire dimostrazioni formali si vuole ora costruire un esempio apposito affinché si possa avere una ragionevole

convinzione circa l'assenza di incongruenze nel lavoro di rielaborazione svolto.

Esempio. Prendendo in considerazione i valori relativi alla versione 5 di livello H, presenti nell'estratto della tabella rappresentato in figura 3.3, abbiamo 134 codewords totali divise in 88 codewords di Ecc (ovvero la ridondanza che supporta la correzione di errore) e 46 codewords di Dati (si ricorda ancora una volta che una codeword nel contesto dello standard Quick Response rappresenta un byte di 8 bits).

Ognuno dei 4 blocchi di Reed-Solomon possiede 22 codewords di Ecc, mentre per quanto riguarda i dati, i primi 2 blocchi posseggono 11 codewords, mentre gli altri 2 ne hanno 12. Al fine di evitare ambiguità si vuole precisare che in questo esempio stiamo riferendoci alle codewords dei dati come le informazioni di interesse da codificare non ridondate, mentre alle codewords di ecc come la ridondanza necessaria per consentire correzione d'errore e quindi affidabilità; questa suddivisione così netta è fatta per analizzare e comprendere la logica sottostante la codifica, tuttavia si vuole sottolineare come, relativamente all'atto finale di rappresentare il totale delle codewords tramite gli elementi a contrasto costitutivi del simbolo, non vi sia alcuna separazione o differenziazione tra le codewords di uno o dell'altro tipo, che saranno di fatto indistinguibili; come a dire che esiste un'unica area nel simbolo per la totalità dei dati ridondati, non sottoaree specifiche separate che minerebbero l'affidabilità della lettura del simbolo stesso.

Tornando al nostro esempio queste semplici constatazioni mostrano le correlazioni tra i valori in tabella:

*4 * 22 = 88 Ecc Codewords*

*11 * 2 + 12 * 2 = 46 Data Codewords*

88 + 46 = 134 Total Codewords

La capacità di correzione d'errore di ciascun blocco è data dall'ultimo valore delle triple dell'ultima colonna di figura 3.3, per cui avremo per i nostri 4 blocchi una capacità di correzione di (al più) 11 * 4 = 44 codewords.

Calcoliamo a questo punto il semplice rapporto tra questo (massimo) ammontare di dati che si è in grado di correggere ed il totale delle codewords: 44/134 = 32,8 %. Infatti il livello H di correzione d'errore dei codici Quick Response stabilisce che la percentuale massima di codewords errate sul totale, tale che possano ancora essere corrette ed il simbolo decodificato, deve poter raggiungere circa il 30% (proprio con il livello H che è il più elevato e dunque quello che fa maggior uso di ridondanza).

Ora che abbiamo ragionato su cosa accade relativamente allo standard Quick Response, vediamo brevemente lo stesso caso con un codice Quick Response esteso al fine di supportare schemi a colori; si capirà così se in virtù delle modifiche proposte otterremo ancora una situazione consistente. Considerando le parametrizzazioni che sono state proposte relativamente alle tabelle dello standard, e ragionando su un'evoluzione dei codici che usi uno schema a 4 colori (dunque 2 bits/module) per una versione 5 di livello H analoga alla precedente, si avranno 268 codewords totali, divise in 176 di Ecc e 92 di Dati. Ognuno degli 8 blocchi di Reed-Solomon avrà 22 codewords di Ecc, mentre per i dati, i primi 4 blocchi avranno 11 codewords, mentre gli altri 4 ne avranno 12.

*8 * 22 = 176 Ecc Codewords*

*11 * 4 + 12 * 4 = 92 Data Codewords*

176 + 92 = 268 Total Codewords

La capacità di correzione d'errore per gli 8 blocchi sarà di 11 * 8 = 88 codewords (il valore 11 si prende ancora una volta dall'ultimo campo della triple poste nella colonna in estrema destra di figura 3.3 che rappresenta la capacità di correzione d'errore di quel dato blocco espressa nel massimo numero di codewords correggibili).

Calcoliamo il rapporto tra il valore trovato per la capacità di correzione d'errore complessiva e le codewords totali in maniera analoga alla precedente, ottenendo: 88/ 268 = 32,8 %. Dunque è facile convincersi, in virtù dei semplici coefficienti moltiplicativi, che i livelli di correzione d'errore L, M, Q ed H nell'evoluzione proposta di codici Quick Response a maggiore capacità vengono preservati, fatto importante per il raggiungimento degli obiettivi discussi in sezione 3.1.

Il Character Count Indicator

Secondo l'esempio trattato in relazione alle tabelle parametrizzate, sembrerebbe che un codice Quick Response esteso con uno schema a diversi colori rimanga ancora coerente, eppure un simbolo Quick Response è un oggetto piuttosto complesso ed indirettamente, lavorando all'estensione delle tabelle, abbiamo creato una incongruenza relativamente alla gestione dei "Modes": non si deve infatti dimenticare di gestire il Character Count Indicator. Questo ha la funzione di esprimere quanti caratteri, in accordo al relativo Mode, sono codificati nel simbolo; coll'estensione proposta potrebbe capitare di dover codificare più caratteri di quanti solitamente una certa versione del simbolo, utilizzando un certo Mode, possa fare. Il valore da esprimere nell'indicatore potrebbe essere troppo grande, qualora siano stati assegnati al Character Count Indicator un numero di bits eccessivamente esiguo.

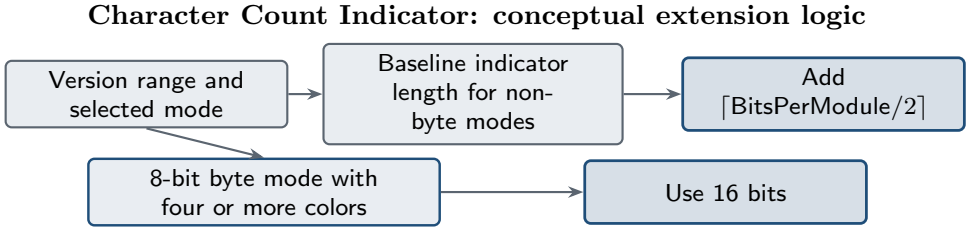

Figura 3.4: Logica concettuale di estensione del Character Count Indicator.
Nota editoriale (2026): La tabella numerica della figura del 2010 non è riprodotta in questa edizione digitale pubblica. Le sue colonne, tutte con sfondo bianco, erano, da sinistra: (1) Version; (2) Numeric Mode; (3) Alphanumeric Mode; (4) 8-bit Byte Mode; (5) Kanji Mode. Le righe raggruppavano le versioni 1–9, 10–26 e 27–40. Il nuovo schema mostra soltanto la logica di estensione; i successivi riferimenti del 2010 ai bit leggibili nella figura riguardano i valori omessi. Riferimento tecnico: ISO/IEC 18004:2000.

In figura 3.4 si possono leggere i bits allocati ad ospitare il character count indicator: come si può notare nella definizione dei valori presentata dallo standard si è cercato di ottimizzare al massimo, fino a cercare di risparmiare perfino il singolo bit. Questa caratteristica, di per sé positiva, pone tuttavia un difetto nel nostro progetto di evoluzione dei codici Quick Response: utilizzando schemi a 4, 8 o 16 colori l'ammontare complessivo di informazione rappresentabile diviene rispettivamente circa il doppio, il triplo o il quadruplo rispetto ad un codice standard, e conseguentemente per il character count indicator subentra la necessità di ottenere almeno 1 o 2 bits aggiuntivi.

Esempio. Per simboli appartenenti a versioni nell'intervallo che va da 1 a 9, scegliendo di utilizzare l' Alphanumeric Mode, al Character Count Indicator sono assegnati 9 bits, con i quali si possono quindi esprimere valori fino a 511. In virtù della maggiore densità di dati mediante l'uso dei colori, questi simboli potrebbero ora dover racchiudere più di 511 caratteri alfanumerici, una situazione che il Character Count Indicator non potrà più gestire.

Si vuole proporre nel seguito una possibile estensione della tabella, sintetizzata

dai seguenti criteri :

- Per quanto riguarda la modalità 8-bit byte Mode la via più semplice è di assegnare direttamente 16 bits indipendentemente dalla versione[2], qualora si stia usando uno schema a 4 o più colori.

- Per quanto riguarda tutte le altre modalità (ad esclusione quindi dell' 8-bits byte mode trattato a parte) si parametrizzino i valori della tabella nella seguente maniera[3]:

  - $CharCountIndicatorLen_{new} = CharCountIndicatorLen_{old} + BitsPerModule/2$

3.3.2 Gestire l'Introduzione dei Colori nei Quick Response

In risposta alla problematica contrassegnata con (2.) in sezione 3.2.3, dobbiamo da un lato cercare di mantenere retro-compatibilità coi codici Quick Response standard, mentre dall'altro ci resta il problema di non sapere come codificare l'informazione di quanti colori stia utilizzando un simbolo relativo alla nuova simbologia (schema a 2, 4, 8, 16 colori?), considerato che è critico che il lettore ne venga in possesso affinché il processo di decodifica possa andare avanti.

In risposta alla problematica contraddistinta da (3.), dobbiamo consentire al lettore di venire a conoscenza di quali colori siano stati effettivamente usati in fase di codifica da parte del generatore; che il decodificatore possa sapere quale schema di colori è stato utilizzato è davvero un punto centrale della questione, dunque resta da ragionare se proporre un accordo tra generatore e lettore, cosicché i colori utilizzati

[2]Per codici Quick Response che usano il classico schema in bianco e nero si continuino ad usare 8 bits per le versioni minori o uguali alla nona quando si è in modalità 8-bit byte mode.

[3]Per valori di BitsPerModule dispari diversi da 1 (quindi escludendo il caso base di Quick Response standard) si approssimi all'intero superiore la quantità BitsPerModule/2 presente nella formula.

siano noti a priori, oppure cercare altre vie. Anche qualora si venga a capo della questione, dobbiamo però aspettarci una certa distorsione cromatica causata dalla fotocamera, di cui non si può escludere l'effetto in quanto sicuramente presente, e che va certamente gestito al fine di ottenere un oggetto, come il Quick Response a colori a maggiore capacità, che sia effettivamente utilizzabile nel mondo reale.

Nel seguito riepiloghiamo un'ultima volta gli interrogativi da affrontare per procedere subito dopo a ragionare sulle possibili soluzioni che possono venirci incontro per una rielaborazione completa dei simboli, che possa poi coronarsi nella realizzazione di lettori predisposti alla loro effettiva decodifica, in particolare relativamente ai dispositivi mobili.

Fermo restando che si vuole mantenere compatibilità coi codici Quick Response standard dobbiamo capire come rappresentare l'informazione di quanti colori (dunque quanti bits per modulo) siano stati usati per la codifica del simbolo. Inoltre questi colori, dovrebbero essere noti a priori al decoder? E come si può pensare che lo scatto della fotocamera non modifichi sensibilmente le sfumature cromatiche che ci aspettiamo?

Soluzioni proposte

Si vuole proporre in questo contesto l'introduzione di un nuovo campo(un nuovo pattern) per l'evoluzione dei codici Quick Response (che non crei conflitti con elementi preesistenti dei codici QR standard), costituito da una tavolozza di colori, utilizzata come legenda.

In questa maniera molti problemi di distorsione cromatica si risolvono alla fonte, poiché se i colori dei moduli dell'area dati+ecc vengono alterati, così avviene anche per

i colori dei moduli della legenda, preservandosi quindi il funzionamento in decodifica (il tutto ovviamente entro certi limiti, poiché se lo scatto fa in modo che i colori si alterino divenendo quasi uguali, il problema rimane insoluto e la lettura del simbolo fallisce). Gli interrogativi che rimangono sono ancora relativi a come codificare il numero di colori, ma anche a capire quando e se convenga far fallire il processo di decodifica e riprovare con un nuovo scatto anziché fare del lavoro inutile, in circostanze di scansioni non ottimali. Seguendo una logica di quick fail, è molto scomodo andare avanti nel processo, quando i colori rilevati sono alterati in maniera da essere scarsamente riconoscibili. Infatti si andrebbe avanti con le computazioni e si perderebbe tempo di cpu, per poi fallire comunque. Un concetto importante da seguire è quindi, di concedere delle alterazioni cromatiche, ma entro certi limiti. Una soluzione semplice e performante può essere di dire che la legenda debba ospitare inizialmente i colori scuri e poi i chiari o viceversa: dando cioè un ordine ai colori che ci aspettiamo, possiamo valutare, se al di là della sfumatura cromatica specifica, la luminosità del colore è quanto meno corretta rispetto a quella attesa: se è così possiamo sicuramente procedere, altrimenti conviene fallire.

Al fine di evitare che la legenda diventi un fastidioso collo di bottiglia che fa fallire spesso l'algoritmo ed allo stesso tempo per garantire maggiore precisione cromatica è una buona idea pensare di replicare la legenda. Qualunque legenda che non rispetti i criteri di luminosità in fase di decodifica dovrebbe essere scartata (se addirittura un modulo colorato scuro viene letto chiaro o viceversa, possiamo pensare che campionare il colore di quel modulo è inammissibile); tra tutte le rimanenti legende che soddisfano il vincolo e dunque ritenute valide, si può invece pensare di fare una media tra le sfumature cromatiche campionate di moduli analoghi. Con questa soluzione svincoliamo comunque il decoder dal dover sapere quali colori aspettarsi: il codificatore ha libero

arbitrio nella scelta dei colori quando crea un codice, unico vincolo un ordinamento per luminosità crescente o decrescente degli stessi. Con questo sistema otteniamo anche un altro vantaggio: il decoder può capire quanti colori siano stati usati senza che sia presente un campo a parte che codifichi questa informazione: può essere sufficiente studiare le legende replicate[4], i cui moduli sono ordinati per luminosità, per capire il numero di colori usato.

La posizione delle legende replicate può essere assegnata in vari punti, ad ogni modo va evitato che comprometta la retro-compatibilità con i codici Qr standard, per cui si può provare a inserirla esternamente al quadrato che costituisce il simbolo oppure cercare di inserirla all'interno dello stesso ma senza rovinare i patterns esistenti. La cosa più pulita sarebbe poter utilizzare per le legende gli Extension Patterns, pensati proprio per future estensioni di funzioni di codici Quick Response. Poter utilizzare campi del genere sarebbe perfetto, tuttavia sono ammessi solo nei codici Quick Response Modello 1, una simbologia sconsigliata, in disuso.

[4]Per esigenze legate all'affidabilità in lettura, al bilanciamento tra aree scure e chiare, anche in ottica di mascheramento, si propone inoltre l'ulteriore vincolo che esattamente metà dei colori della legenda sia chiara e metà dei colori sia scura, concetto che verrà chiarito meglio in seguito.

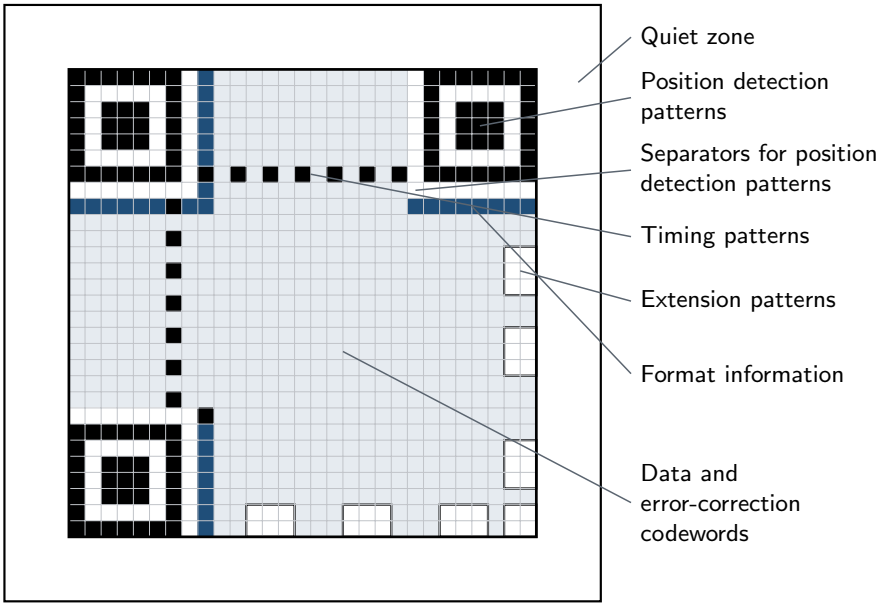

Figura 3.5: Schema creato dall'autore delle regioni funzionali di un simbolo Model 1.
Nota editoriale (2026): Questo schema è stato creato dall'autore per l'edizione digitale e non riproduce una figura dello standard; riferimento tecnico per la discussione: ISO/IEC 18004:2000.

Nel nostro caso, come prima proposta, le legende saranno giustapposte al quadrato che rappresenta il codice Qr, quindi si troveranno esternamente ad esso. Ci si riserva di elaborare un posizionamento più accurato in futuro, ma la soluzione in questione è già sufficiente per ottenere un primo oggetto sufficientemente affidabile.

3.3.3 Reinventare il concetto di Data Masking

Per quanto riguarda il concetto di data masking e dunque ragionando circa le otto maschere a disposizione nello standard Quick Response, ricordiamo che il loro obiettivo centrale è quello di aumentare l'affidabilità del codice, più precisamente agevolare il processo di lettura ottica garantendo da un lato che il simbolo non sia illeggibile, dall'altro supportando una rappresentazione ottica dei dati che sia a più alto contrasto.

Un esempio relativo a quest'ultima funzione, ovvero aumentare il contrasto degli elementi costitutivi del simbolo, è dato dal fatto che le maschere consentono un im-

portante bilanciamento delle aree chiare e scure, evitando il formarsi di simboli con rapporti tra le superfici chiari e scure che si allontanino troppo dalle proporzioni bilanciate 50%-50%; relativamente al loro ruolo di impedire la formazione di codici illeggibili evitano ad esempio il formarsi di aree simili ai position detection patterns all'interno dell'area dati+ecc (può capitare per caso in alcune rare istanze di codici, le quali prive di mascheramento, diverrebbero illeggibili e prive di qualsiasi utilità).

Tenendo presente queste considerazioni, appare chiaro che dobbiamo considerare la luminosità dei moduli anche in un eventuale codice Quick Response che possieda moduli di varie sfumature cromatiche (si parla di moduli a colori sempre e solo per l'area dati+ecc). Se così non fosse, se non considerassimo la luminosità dei moduli colorati, se non rielaborassimo il concetto di mascheramento, potrebbero ad esempio formarsi aree nell'area dati che per luminosità simulano i position detection patterns, trovandoci di fronte a codici de facto illeggibili.

In risposta alla problematica contrassegnata con (4.) in sezione 3.2.3, vogliamo capire come procedere col mascheramento, dal momento che, evolvendo i codici, si ha ora che ad un modulo possono essere associati più bits, fatto non contemplato nello standard ovviamente. Inoltre se normalmente, un modulo nero viene invertito con uno bianco e viceversa, in uno schema a più colori dobbiamo proprio ridefinire come sarebbe opportuno procedere, come ad esempio capire quali colori vanno invertiti con quali altri colori.

Cerchiamo ora di proporre delle soluzioni che siano applicabili e che non richiedano di dover sconvolgere la struttura del procedimento di Masking, ma puntino piuttosto a riadattarlo al nostro scenario.

Si consideri un codice Quick Response a colori come una binary matrix, in cui per la posizione di ogni modulo sia valutato se questo sia o meno luminoso (chiaro o scuro).

Su una tale matrice si può ancora eseguire il tradizionale algoritmo per valutare la bontà delle 8 maschere e scegliere la migliore tra queste per i passi successivi. Quindi ricapitolando, si sta proponendo di partire da un codice Quick Response a colori non mascherato, generare a partire da questo un simbolo che veda mappati i moduli colorati in moduli chiari e scuri a seconda delle luminosità dei vari colori, ed infine utilizzare questo codice prodotto ad hoc come simbolo sul quale valutare la migliore maschera per il simbolo a colori di partenza che abbiamo l'esigenza di mascherare.

Ora che si è detto come scegliere la migliore tra le maschere a disposizione, una volta generato tale mask pattern come si può procedere ad effettuare le operazioni di xor, quando ogni cella rappresenta più di un bit per modulo? Inoltre, considerando che un modulo possiede un determinato colore nell'area dati+ecc di questa nuova simbologia basata sui Quick Response, che significato potrebbe avere "invertire" tale colore? Relativamente allo standard questo concetto è ovvio, ovvero si possono invertire celle bianche con celle nere e viceversa, mentre nel nostro contesto evidentemente il procedimento va ripensato.

Per quanto riguarda il primo punto si può eseguire semplicemente lo xor tra ogni coppia (i-esimo bit del modulo, valore del mask pattern), ottenendo quindi per ogni sequenza di bits associata al modulo, per effetto del mascheramento, o che questa rimarrà la medesima, oppure che ogni bit della sequenza venga invertito.

Esempio. Se abbiamo un Quick response che usi uno schema a 4 colori, e consideriamo un modulo che rappresenti la sequenza "01", dopo il mascheramento così per come è stato definito si avrà ancora "01", oppure "10". Nel primo caso la maschera chiede che, qualora il modulo fosse chiaro rimanga tale o se scuro idem, nel secondo caso chiede che ne sia "invertita" la luminosità. In che maniera può essere però inver-

tita tale luminosità, il modulo colorato come va invertito? Si è detto che è qualcosa di non ben definito.

A seguito delle varie osservazioni fatte, cercando di combinarle per trovare una soluzione che consenta di centrare gli obiettivi che ci siamo prefissati, si può sintetizzare la seguente proposta:

- Ogni legenda abbia la prima metà di moduli contenenti colori "chiari", la seconda metà di moduli contenenti colori "scuri" o viceversa (50% e 50% supporta un migliore bilanciamento di luminosità).

- La sequenza di bits associata ad ogni colore sia tale che qualora ogni suo bit sia invertito, si ottenga la sequenza di bits appartenente ad un colore che sia di luminosità opposta al primo.

Per una maggiore comprensione, a titolo di esempio, mostriamo in figura 3.6 la struttura del procedimento di Data Masking, basata sull'operazione di xor, relativa allo standard Quick Response. Si noti il mapping tra i valori assunti dai bits della matrice, nonché la colorazione basata sul classico schema in bianco e nero che ne consegue relativamente ai moduli costitutivi del simbolo.

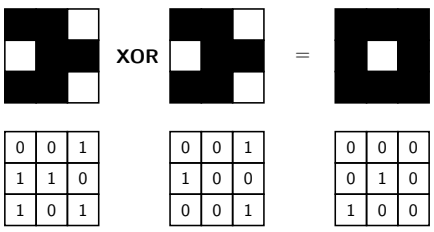

Figura 3.6: Esempio di Data Masking in accordo allo standard Quick Response
Nota editoriale (2026): Questo esempio è stato creato indipendentemente dall'autore per l'edizione digitale per illustrare la spiegazione del 2010; non riproduce una figura dello standard.

La figura 3.7 mette invece in evidenza come è stato ripensato il mascheramento dei dati per adattarsi alle nuove esigenze nate dal tentativo di introduzione dei colori

come veicolo ulteriore di informazione in relazione all'uso di codici bidimensionali. Si faccia il confronto diretto con figura 3.6 e si vedrà che si è cercato di evolvere un sistema performante e maturo, come il procedimento di data masking definito dallo standard, nella maniera più diretta possibile.

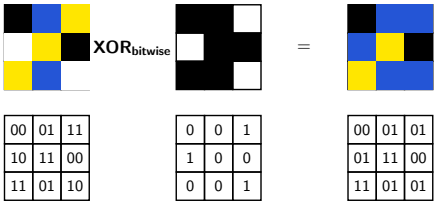

Figura 3.7: Esempio di Data Masking rielaborato per la nuova simbologia
Nota editoriale (2026): Questo esempio è stato creato indipendentemente dall'autore per l'edizione digitale per illustrare la proposta del 2010; non riproduce una figura dello standard.

Con questi ultimi esempi chiudiamo il quadro relativo al Data Masking, di cui si è discusso finora, tuttavia si vuole concentrare l'attenzione su un fattore collaterale al mascheramento, che però è fondamentale gestire. Infatti, siamo arrivati a capire come invertire un modulo colorato dicendo che si può prendere come opposto quel colore che rappresenti una sequenza di bits tale che se di cambia ogni singolo bit si riottiene la sequenza associata al primo colore, quello da invertire. Si è inoltre detto che queste coppie di colori "opposti" al livello di sequenza di bits rappresentata, debbano anche essere opposti quanto a rispettive luminosità.

Pur ragionato su tutto questo, non si è espressa alcuna opinione su come organizzare questi colori in una legenda, come scegliere gli "opposti", in sostanza quindi in che maniera ordinare le varie sfumature cromatiche. A questo proposito si valutano due possibili alternative, al fine di giustificare la soluzione che è stata poi presa come riferimento.

Una prima possibilità è una ordimento totale della legenda per luminosità, cosa che ancora rispetta il vincolo di avere colori luminosi e colori scuri suddivisi nelle

rispettive metà, per cui è una soluzione valida, che è stata inoltre testata e sperimentata funzionante. Possiamo tuttavia affermare che tale scelta tenda a marcare il contrasto, fenomeno utile per incrementare l'affidabilità del codice?

Se guardiamo in figura 3.8, all'area di sinistra corrisponde proprio questa configurazione, che rappresenta un ordinamento totale dei colori per luminosità. Si possono notare i mapping tra le sequenze di bits ed i colori, il loro ordinamento, nonché un tabella che analizza il contrasto tra i moduli relativamente a questa prima maniera di strutturare la legenda. Si noti a questo proposito, la non convincente prestazione di mascheramento che si ottiene tra i colori centrali, che benché "opposti", in quanto appartenenti ai due diversi gruppi di luminosità, hanno in realtà una differenza intrinseca di luminosità estremamente bassa: Dunque, guardando l'esempio in figura, può sorgere il dubbio che non sia eccessivamente performante invertire un modulo che rappresenti la sequenza "011", rappresentata da un modulo di un colore caratterizzato da una luminosità pari a 0.75, con la sequenza "100", relativa ad un modulo di un altro colore che abbia una luminosità pari a 0.65.

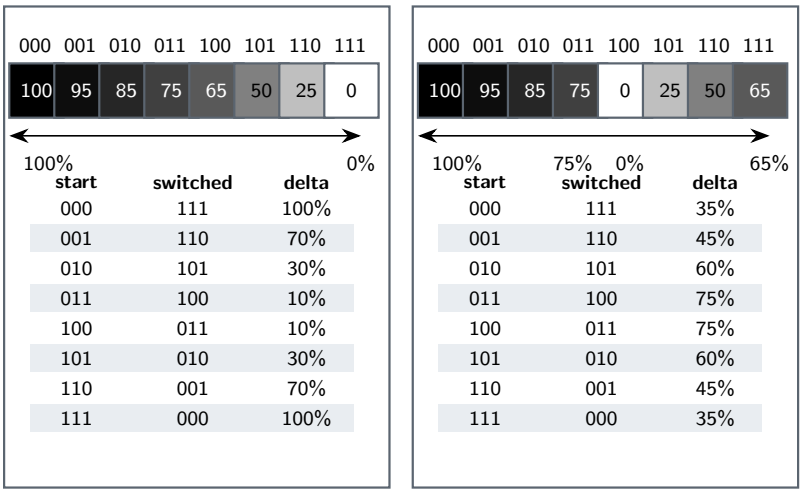

Figura 3.8: Soluzioni alternative per l'organizzazione interna dei moduli colorati delle legende. Ai colori effettivi sono sostituite in figura le rispettive luminosità.

Nota editoriale (2026): Questo confronto vettoriale è stato ricomposto indipendentemente dall'autore per l'edizione digitale a partire dai valori e dalle relazioni esposti nel testo del 2010.

Per questo motivo viene proposta una seconda alternativa, espressa nella parte destra di figura 3.8, che anziché mantenere alto contrasto tra i moduli agli estremi e basso contrasto tra i moduli centrali come avveniva nel caso precedente, cerca di spalmare equamente le differenze di luminosità, garantendo un contrasto sufficiente per ogni coppia di moduli opposti.

3.3.4 Definire una Metrica di Distanza Cromatica

Relativamente alla problematica contrassegnata con (5.) in sezione 3.2.3, dovendo riconoscere le diverse sfumature cromatiche, abbiamo necessità di una metrica di distanza tra i diversi colori. La precisione nel riconoscere colori "simili" in un'immagine è critica tanto da precludere ogni possibilità di decodifica del simbolo, qualora non si raggiunga un'accuratezza sufficiente; fatto che sancirebbe il fallimento del progetto di evoluzione dei codici Quick Response.

Fortunatamente anche con una metrica di similarità piuttosto semplice e immatura come quella che è stata presa come riferimento per le prime sperimentazioni si può centrare l'obiettivo di raggiungimento di un'accuratezza tale da supportare schemi a più colori.

Una metrica di similarità abbastanza intuitiva può essere infatti considerata la distanza euclidea sia nello spazio tridimensionale "rgb" (red-green-blue), che nello spazio tridimensionale "YCbCr" (luminanza-crominanza blu-crominanza rossa). In virtù della sua semplicità, non genera prestazioni eccellenti, ma senz'altro non richiede grande sforzo computazionale. Nonostante come lavoro futuro ci si riservi di lavorare alla definizione e all'adozione di una metrica di distanza cromatica di superiore qualità, questa semplice metrica consente un primo utilizzo dei codici, riuscendo a consentire la definizione di un valido concetto di "similarità" tra i colori.

Capitolo 4

Progetto e Realizzazione di Generatore e Lettore

In questo capitolo si tratteranno la realizzazioni sia di un generatore, sia di un lettore, associate all'utilizzo di codici Quick Response a colori ad alta capacità [4], codici sui quali si è discusso ampiamente nel capitolo precedente. Poiché la simbologia di codici trattata dovrà essere utilizzabile tanto in ambito desktop quanto su dispositivi mobili, sarà introdotto il sistema operativo Android O.S., correntemente in uso su una vasta gamma di smartphones di ultima generazione, che è stato scelto come piattaforma per lo sviluppo del lettore in ambiente mobile. Una volta trattate le implementazioni suddette, si potranno fornire i primi risultati sperimentali nei capitoli successivi.

L'evoluzione dei codici Quick Response che è stata presentata nel capitolo precedente è stata realizzata e sperimentata valida. Infatti, fermo restando l'utilità di ulteriori miglioramenti e ottimizzazioni tanto negli algoritmi di image processing, quanto nella rielaborazione delle componenti dei simboli Quick Response, si è comunque prodotta una prima versione delle applicazioni di codificatore e di lettore, in grado di generare, riconoscere e decodificare con successo la nuova simbologia di codici bidimensionali che trae le basi dall'esperienza maturata dallo standard ISO/IEC 18004 relativo ai codici Quick Response.

4.1 I Progetti di Riferimento

Sarebbe stato difficile portare avanti rapidamente la realizzazione di generatori e lettori per i nuovi codici, senza avvalersi degli ottimi risultati già ottenuti da progetti pre-esistenti, in particolar modo per tutti gli algoritmi di processamento dell'immagine per il riconoscimento dei simboli, un task ritenuto piuttosto impegnativo. Per quanto riguarda lo sviluppo di un generatore per i codici Quick Response a colori, ci siamo basati sulla libreria opensource libqrencode[1] [5], grazie alla quale si è prodotto un applicativo che soddisfa le nostre esigenze in maniera abbastanza completa. Il lavoro futuro in questo ambito può essere al più focalizzato a migliorare la struttura del codice o ad apportare fixes. La realizzazione di un lettore deve ringraziare la libreria opensource ZXing[2] [6], un progetto che ad oggi prosegue nell'affinare tecniche di riconoscimento di diverse simbologie di codici, tra i quali i Quick Response di nostro interesse. La caratteristica più interessante del progetto ZXing è probabilmente l'estrema portabilità, essendo un progetto che sebbene abbia come obiettivo principale lo sviluppo di lettori di codici per la piattaforma Android, non disdegna l'idea di mettere insieme porting dell'applicazione che vadano su Iphone O.S. o che semplicemente siano disponibili su parte della gamma di cellulari che supportano la piattaforma Java Micro Edition[3]. Il lavoro futuro in ambito di decodifica sembra più ricco di sfide concettuali rispetto al codificatore, poiché è qui che si concentra tutto il lavoro di processamento dell'immagine, e dunque le sfide nel rilevare i codici, nel gestire distorsioni, nel campionare correttamente la posizione dei moduli, volendo ora in aggiunta distinguerne anche le diverse sfumature cromatiche.

[1] link: http://megaui.net/fukuchi/works/qrencode/index.en.html
[2] link: http://code.google.com/p/zxing/
[3] Si vuole sottolineare che tali porting dell'applicativo non raggiungono tuttavia al momento la qualità vista su piattaforma Android.

4.2 Introduzione ad Android O.S.

Android O.S. è un sistema operativo opensource per dispositivi mobili che, basato su kernel linux e diffusosi solo di recente, presenta potenzialità di espansione enormi per il prossimo futuro. Gestito dalla cosiddetta Open Handset Alliance, Android è de facto la piattaforma promossa dal suo membro predominante, ovvero Google. L'alleanza incorpora membri come Htc, Lg o Motorola che si occupano sostanzialmente della produzione dell'hardware, come anche T-Mobile o Vodafone per quanto riguarda gli operatori mobili, ma la produzione del software, ovvero lo sviluppo del sistema Android, che è il punto di nostro interesse, è gestita quasi totalmente da Google (nonostante nell'alleanza siano presenti altri produttori software minori).

Android O.S. nasce senz'altro per tentare di entrare in maniera competitiva in un mercato in cui la Apple, produttrice dell'iPhone, negli ultimi anni ha sicuramente dominato. Portare vie fette di mercato ad un prodotto come l'iPhone, già molto diffuso e ben radicato nel mercato, a fronte dell'ampio consenso di cui gode, con utenti oltremodo soddisfatti, non è certo una sfida da poco conto.

Un utente che sceglie Android tuttavia riesce, vista la vasta gamma di smartphones che lo adoperano, a trovare anche prodotti di fascia economica oltre a quelli di fascia alta, cosa che potrebbe consentire alla Open Handset Alliance di assorbire un bacino di utenti che, per via dei costi maggiori, non può permettersi iPhone.

La maturità di un prodotto come iPhone, che continua comunque ad evolvere di anno in anno, propone all'utente un'esperienza che non è certamente inferiore a quella che può fornire l'Android di oggi. Il fatto è che Android 1.5 e 1.6, ovvero quelle che sono tuttora le più diffuse versioni della piattaforma, non sono distribuzioni del software ancora mature; sono già stati rilasciati i firmware 2.1 e 2.2 tuttavia a metà

2010 sono ancora scarsamente operative sulla gran parte dei dispositivi, in quanto il sistema di aggiornamento del software soffre di carenze strutturali: nonostante nuove versioni della piattaforma Android siano costantemente rilasciate, l'aggiornamento deve passare per le mani del produttore hardware, che può decidere di non rilasciarlo; per fare un esempio Htc ha già dichiarato che non aggiornerà a Froyo (2.2) nessuno degli smartphones venduti nel 2009 come Htc Magic o Htc Hero.

Android è tuttavia un sistema aperto, se l'aggiornamento non è rilasciato dal produttore hardware, teams liberi di sviluppatori non potrebbero farlo in maniera non ufficiale? Il punto è che il sistema è a sorgente aperta ma solo la casa produttrice possiede codice maturo per i drivers dei componenti hardware (come fotocamera, accelerometro, etc...) che lei stessa ha prodotto. Il risultato è che distribuzioni non ufficiali, per quanto impegno venga messo nella produzione del software, avranno sempre l'inconveniente di trovarsi in difficoltà nel rendere il dispositivo completamente funzionante a livello di ogni singolo componente hardware.

Nonostante quanto si è detto circa alcune lacune, Android possiede caratteristiche davvero interessanti che ci hanno avvicinato all'utilizzo di tale piattaforma sia come utenti che come sviluppatori. In primis, il fatto di essere un sistema aperto è un pregio da non sottovalutare, ben diverso da quanto avviene con le tecnologie Apple, tutte strettamente proprietarie; in secondo luogo, il sistema Android potrà raggiungere presto la maturità del software, nonché estendere rapidamente la propria diffusione in tutto il mondo, vista l'importanza delle società che lo promuovono (Google in primo piano). Ci sono poi altri vantaggi se guardiamo la questione dal punto di vista dello sviluppatore, come ad esempio il fatto che il software development kit per lo sviluppo di applicazioni destinate ad Android è stato rilasciato per Windows, per Linux e per Mac O.S., venendo incontro alle esigenze di ogni programmatore, mentre Apple

limita la realizzazione di applicazioni per iPhone ai soli sviluppatori che comprino i suoi macintosh, precludendo l'sdk alle altre piattaforme.

In conclusione, se si è scelto di lavorare su Android non è un caso, in quanto lo si ritiene l'unica valida alternativa ad iPhone O.S., a differenza ad esempio di altri sistemi come windows mobile.

4.3 Impatto di Android O.S.

Dopo aver fatto il quadro della situazione sulla piattaforma di sviluppo del nostro progetto, seppur senza scendere nello specifico, possiamo passare a ragionare su un argomento più mirato al lavoro di tesi, ovvero trattare l'impatto che avrà Android sul progetto di realizzazione di un lettore per la nuova simbologia di codici, in virtù della sua funzione di software di base dei dispositivi mobili, ovvero di piattaforma che dovrà fornire quelle api di cui necessiterà la nostra applicazione.

4.3.1 Limitazioni di Portabilità del Codice

Dal punto di vista dello sviluppatore, lavorare su Android richiede l'utilizzo del linguaggio Java; se poi si hanno anche esigenze prestazionali stringenti si possono scrivere porzioni di codice in C da compilare come librerie native. Alla base del sistema c'è un kernel linux quindi potremmo aspettarci, per quanto si è detto, di avere a disposizione da un lato la Gnu C Library, per quanto riguarda la programmazione in C, dall'altro la Java Standard Edition o la Java Micro Edition a supporto del nostro codice Java. Purtroppo la situazione non è questa, in quanto pur essendo la piattaforma Android un sistema basato su linux, non è una distribuzione linux: si può utilizzare solo una parte del set di librerie GNU, fatto che rende scarsamente portabile codice originariamente scritto per linux. Se si considera inoltre che Android è volutamente privato del

supporto sia della Java Standard Edition, che della Java Micro Edition, ecco che di fatto è annullata la portabilità di codice pensato originariamente per altri ambienti.

4.3.2 Un Emulatore Carente per il Progetto

Il progetto di sviluppo di un lettore per i nostri codici richiede ovviamente che sui dispositivi di interesse siano presenti, a livello hardware, dei sensori ottici. Android consente l'accesso alla fotocamera da parte delle applicazioni, tuttavia l'emulatore fornito con il software development kit, è carente di tale supporto. Sembra scontato che per sviluppare per un dato dispositivo sia necessario possederlo fisicamente, tuttavia una buona parte delle applicazioni per Android possono essere realizzate con il solo uso dell'emulatore, nonostante non sia il caso del nostro progetto, che ha una necessità purtroppo non supportata. Non è in realtà tanto l'inconveniente di dover possedere un dispositivo reale a creare difficoltà, ma le conseguenze che questo comporta: ogni sviluppatore sa bene quanto è importante avere a disposizione un ambiente che consenta rapide esecuzioni dell'applicazione su cui si sta lavorando, mentre per il progetto di realizzazione del lettore ottico, ogni volta che si vuole fare un test si deve rilanciare l'installazione dell'applicativo sul dispositivo per poi mandarlo in esecuzione; un'operazione che nel complesso è molto più lenta di quanto avviene lavorando in ambiente desktop. In questo contesto, per completare il quadro su cosa comporti lavorare in tale ambiente, va detto che vi è probabilmente una maggiore difficoltà di diagnostica e debugging, rispetto a quanto non avvenga in ambiente desktop.

4.3.3 Uso della Fotocamera: Rischi per il Progetto e Considerazioni sullo Sviluppo

Si è detto nel corso della trattazione che le fotocamere poste sui moderni smartphones sono di buona qualità, essendo sufficientemente accurate da poter esser senz'altro

utilizzate per la lettura ottica dei codici. A questo punto tuttavia ci si deve scontrare con problematiche relative da una parte all'uso pratico dall'altra con le caratteristiche dei formati immagine predefiniti.

Per quanto riguarda la prima osservazione, l'utilizzo pratico di un utente che voglia leggere un codice prevede una logica, se vogliamo, a più tentativi; l'utente si aspetta di poter "passare" il dispositivo sul codice finché non avvenga che il lettore riconosca il simbolo e presenti all'utente il risultato del processo di decodifica. Questo implicitamente richiede che il lettore agisca non su di un solo input, bensì su diversi scatti, fino a trovare quello idoneo per il recupero delle informazioni del codice. Ecco perché in realtà il progetto di riferimento Zxing[4], agisce in modalità Camera Preview, facendo in modo che l'utente sperimenti un dispositivo in modalità video, potendo vedere sullo schermo se il codice sta venendo inquadrato correttamente. Quindi per il progetto di evoluzione dei codici Quick Response, possiamo aspettarci di ottenere diversi frames appartenenti ad una sorta di video; ogni frame, ogni scatto, può costituire una diversa occasione di riuscire a decodificare con successo il simbolo. Se si considera che le tali fotocamere sono in grado di produrre immagini di ottima qualità, ad alta risoluzione, non sembrano esserci grandi problemi per il nostro lavoro, tuttavia è presente un'insidia da non sottovalutare: in modalità Camera Preview la qualità di ogni singolo scatto è abbastanza bassa e cosa più importante il formato predefinito di Android per tale modalità è molto povera di informazione cromatica, in quanto sottocampiona il colore. Prima di andare nello specifico sui dettagli relativi al sottocampionamento, si vuole far notare che tale formato, precisamente il YCbCr 4:2:0, non è necessariamente l'unico disponibile, tuttavia è l'unico che Android stabilisce che debba essere supportato da ogni dispositivo di tale piattaforma. Per un motivo così fondamentale,

[4]Si veda sezione 4.1 relativa ai progetti opensource di riferimento.

ovvero essere sicuri che il nostro lettore funzioni su ogni dispositivo Android e non solo su pochi smartphones, è necessario che il lettore riesca a lavorare su immagini in formato YCbCr:420 semiplanare.

YCbCr è un termine relativo ad una famiglia di spazi cromatici utilizzata nei sistemi video e di fotografia digitale. Si tratta di spazi tridimensionali, come anche avviene in rgb, ma anziché avere le componenti di base di rosso, verde e blu, si consideranno luminanza, crominanza blu e crominanza rossa, dove la luminanza esprime la luminosità del colore in questione. Nel nostro caso in realtà parliamo di YCbCr 4:2:0 in cui ogni blocco di 2x2 pixels è rappresentato da 4 campioni di luminosità, 1 per ogni pixel, mentre per quanto riguarda la crominanza i 4 pixels condividono un solo campione di tonalità blu ed un solo campione di tonalità rossa. Questo è il sottocampionamento delle componenti cromatiche, una insidia non da poco per le prestazioni del lettore che vogliamo realizzare; resta da spiegare il perché viene impoverita la qualità del colore. La spiegazione è molto semplice e legata alla nostra natura: l'occhio umano è molto sensibile alla componente luminosa e riesce a discriminare anche piccole variazioni della stessa, tuttavia non ha la stessa capacità discriminante per quanto riguarda le componenti cromatiche. Non c'è quindi da stupirsi se i sistemi di compressione video tendono a risparmiare spazio eliminando parte delle informazioni contenute nelle componenti cromatiche, la cui utilità, al fine di una corretta visione, è abbastanza marginale.

In questa rapida panoramica, relativa agli spazi cromatici contrassegnati da YCbCr, rimane da definire il concetto di struttura planare o semiplanare, dal momento che quest'ultima è usata da Android. Una spiegazione molto semplice può essere data nel seguente modo: anziché interlacciare le tre componenti per ogni pixel generando un unico blocco di dati, si possono mantenere separate le informazioni di

luminosità, crominanza rossa e blu, generando tre matrici di dati, ognuna relativa ad ogni "piano" (da qui il termine planare); questi piani se fusi, se sovrapposti, danno poi luogo all'immagine finale. La struttura semiplanare rappresenta una situazione ibrida, la quale nel nostro contesto, ragionando su YCbCr semiplanare, significherà avere completamente separate le informazioni di luminanza da quelle di crominanza, tuttavia le crominanze rosse e blu non formano due piani distinti, ma sono concentrare in un unico blocco, che vede sequenze di informazioni di rosso e blu interlacciate.

Si sono date informazioni specifiche riguardo il formato compresso YCbCr 4:2:0 semiplanare, che essendo il predefinito e l'unico necessariamente presente relativamente alla modalità Camera Preview di Android, diventa anche il nostro formato di riferimento per il progetto; per completezza, finiamo il ragionamento spiegando in quanto consiste il risparmio introdotto dal tale formato compresso, dal momento che ne paghiamo le conseguenze in termini di più bassa qualità. In generale, si può dire che occorrono un ammontare di dati per rappresentare l'immagine digitale pari all'esatta metà di quanti ne servirebbero se si facesse a meno del sottocampionamento. Faremmo volentieri a meno di questo risparmio visto che il lettore che si vuole realizzare ha senz'altro bisogno di quanta più qualità cromatica sia possibile ottenere, ma non è qualcosa che dipende da noi.

Tornando a ragionare relativamente ad Android, è curioso come il supporto di api per lo sviluppo di applicazioni che sfruttino i dati relativi ai frames catturati in Camera Preview sia così scarso. Non sono presenti classi per la gestione dei dati ed allo sviluppatore viene fornito soltanto un array di dati grezzo che possiede la caratteristica di rispondere al formato indicato dalla costante YCbCr_420_SP. Inoltre, nonostante la dicitura YCbCr (Luminance-Blue-Red) i dati relativi a crominanza rossa e blu sono invertiti, fatto di cui non si era fatto cenno. In pratica, è stato necessario capire

come utilizzare questi dati senza un supporto ed una documentazione adeguata, in più sarebbe stato di grande aiuto avere a disposizione almeno qualche traccia su come, in caso di necessità, convertire il blocco di dati fornito in formato rgb.

In realtà in questi giorni, a lavoro di tesi ultimato, è uscita una nuova distribuzione, ovvero Froyo, che sembra cominciare a gestire le carenze finora affrontate; non a caso una nota relativa alle migliorie introdotte recita così: *"New YUV image format API to enable compression from YUV to JPEG and manipulation of YUV data"*. La manipolazione dei dati in formato YUV è esattamente quanto ci premeva, si vuole qui intendere infatti YCbCr, visto che per un abuso di notazione spesso i due termini vengono usati indistintamente. Per gli anni a venire, è sicuramente interessante vedere inseriti già da adesso nuovi supporti relativi alla gestione dei formati di immagini digitali su Android, tuttavia relativamente allo stato attuale e al prossimo futuro, se si considera che più della metà dei dispositivi in commercio eseguono Android 1.5/1.6, un'altra buona parte la versione 2.1 ed al momento la versione 2.2 non si è ancora per nulla diffusa, appare evidente che per uno sviluppatore la miglior cosa rimane programmare con il set minimale di api fornito dai firmware 1.5 e 1.6 (api level 3 e 4), che seppur richieda maggior sforzo nello sviluppo, vista la minore offerta di interfacce da parte della piattaforma, consente poi l'installazione dell'applicazione su tutti i dispositivi Android, anziché solo sugli ultimi modelli.

Ovviamente tutto il discorso fatto vale solo nel futuro la situazione rimane come quella dello stato attuale, ad esempio basterebbe superare le carenze strutturali del processo di aggiornamento dei dispositivi, sostituendo le versioni precedenti di Android con le ultime uscite, per spostare tutto il ragionamento: se tutti gli utenti possedessero smartphones aggiornati, sicuramente non avrebbe più senso limitarsi a programmare con il set minimale di api per favorire la portabilità.

Per ovviare al problema di manipolazione dei dati forniti relativamente all'immagine della fotocamera, si può andare a cercare nei moduli interni di Android, scritti in codice C, i quali nonostante rappresentino codice non invocabile dalle applicazioni, possono essere usati per prendere spunto e produrre una propria funzione in java, che seppure meno performante in quanto interpretata, raggiunge lo scopo d'interesse.

Ragionando su un file[5] sorgente interno della piattaforma Android, è stata possibile ad esempio produrre una funzione di conversione da formato YCbCr 4:2:0 Sp a Rgb. Si noti che per il lavoro di lettura dei codici non è strettamente necessario convertire tutta l'immagine da un formato all'altro, ma è utile avere in caso di necessità la possibilità di manipolare i dati grezzi che rappresentano lo scatto della fotocamera.

Al solo scopo illustrativo segue un estratto di codice che rappresenta la funzione Java che è stata utilizzata per ottenere i dati di uno specifico pixel in rgb, scritta guardando al sorgente C a cui si è accennato.

```
private RgbPixel getArgbPixel(int x, int y){

        int luminance = (0xff & ((int) yCbCr420SPdata[(topOffset + y) *
            width + leftOffset + x])) - 16;
        if (luminance < 0) luminance = 0;

        int offset = framesize + (width * ((topOffset+y)>>1)) +
            (((leftOffset+x)>>1)<<1) ;

        /* L'ordine U/V (Cb/Cr)... sembra invertito in Android?!
        *  I dati interlacciati di crominanza rappresentano
        *  prima il rosso e poi il blu. */

        int u = (0xff & yCbCr420SPdata[offset+1]) - 128;
        int v = (0xff & yCbCr420SPdata[offset]) - 128;

        int luminance1192 = luminance * 1192;

        int r = (luminance1192 + 1634 * v);
        int g = (luminance1192 - 833 * v - 400 * u);
```

[5]Path del sorgente: platform/hardware/msm7k.git/yuv420sp2rg/yuv420sp2rgb.c

```
        int b = (luminance1192 + 2066 * u);

        if (r < 0) r = 0; else if (r > 262143) r = 262143;
        if (g < 0) g = 0; else if (g > 262143) g = 262143;
        if (b < 0) b = 0; else if (b > 262143) b = 262143;

        int rgb = 0xff000000 | ((r << 6) & 0xff0000) | ((g >> 2) &
            0xff00) | ((b >> 10) & 0xff);


        return new RgbPixel(rgb);
}
```

Nell'estratto di codice soprastante, yCbCr420SPdata rappresenta l'array di dati grezzi ottenuti dallo scatto della fotocamera e ricevuti da Android tramite le api relative, mentre framesize rappresenta il numero di pixels totali che costituiscono l'immagine; molti valori numerici che si trovano nel codice sono semplici costanti di conversione a Rgb.

4.4 Realizzazione del Generatore

Il generatore di codici Quick Response a colori è stato sviluppato rielaborando il codice del progetto opensource libqrencode come si è già accennato in sezione 4.1; grazie a questa libreria a sorgente aperto si è potuto sviluppare un codificatore in tempi più brevi di quanto sarebbe stato altrimenti necessario.

Il codice del progetto libqrencode è stato prodotto per l'ambiente linux e scritto in C, di conseguenza anche la nostra rielaborazione continua ad essere di base compilabile solo in tale ambiente; si è però da subito verificata la possibilità di produrre eseguibili per windows (.exe) grazie a CygWin, che introduce una sorta di ambiente linux su windows.

Quindi, mentre per l'ambiente desktop sono stati immediatamente superati i vincoli di piattaforma, per l'ambiente mobile si è dovuto studiare l'Android NDK (Native

Development Kit), che consente di produrre librerie native per Android a partire da sorgenti C, compilabili però con il solo supporto parziale, non totale, della Gnu C Library che viene offerto da Android.

Questo insieme minimale di librerie a supporto della programmazione, tramite ndk, di codice nativo per Android è stato comunque sufficiente per riuscire ad ottenere un porting[6] del generatore che sia eseguibile su Android; la codifica di simboli Quick Response a colori ad alta capacità può quindi essere portata avanti sia in ambiente desktop che su quei dispositivi mobili che eseguono Android.

Il generatore prodotto può quindi ottenere anche quel vantaggio in termini prestazionali che è tipico del codice compilato nei confronti di istruzioni interpretate caratteristiche di linguaggi ad alto livello come Java.

4.4.1 Struttura dei Sorgenti

Il progetto libqrencode è di taglia relativamente piccola, in quanto consta di soli 8 sorgenti (*.c), oltre ovviamente i relativi header files (*.h).

Si andranno ora a descrivere brevemente i vari moduli.

- bitstream.c: Semplice modulo di gestione degli streams di bits. In programmazione, per lavorare a livello del singolo bit, manipolando l'informazione ad una granularità così fine, è necessario utilizzare delle maschere di bits. In questo modulo viene definita la struttura bitstream, nonché le procedure associate alla sua gestione.

```
typedef struct {
        int length;
        unsigned char *data;
} BitStream;
```

[6] Si tratta quindi di codice compilato per l'architettura di processori Arm, anziché per la famiglia x86 così comune in ambiente desktop.

Una struttura molto semplice costituita da un puntatore ad un array di caratteri e da un intero che rappresenti la lunghezza relativa è più che sufficiente in questo caso.

- mask.c: Il modulo si occupa di gestire l'attività di Masking dei codici Quick Response, quindi in primis procedure per la produzione delle 8 maschere, dopodiché per la scelta del mask pattern più idoneo relativamente al simbolo in questione, ovvero di quella maschera grazie alla quale si produce un simbolo che possieda caratteristiche di maggiore affidabilità possibile. Nel seguito si mostra l'estratto relativo proprio all'analisi del simbolo prodotto, il quale ogni volta che viene mascherato con un diverso Mask Pattern va valutato in base a precisi fattori di "demerito", per poter scegliere infine quello che presenta le minori caratteristiche negative.

```
1  __STATIC int Mask_evaluateSymbol(int width, unsigned char *frame)
2  {
3  int x, y;
4  unsigned char *p;
5  unsigned char b22, w22;
6  int head;
7  int demerit = 0;
8
9  p = frame;
10 for(y=0; y<width; y++) {
11         head = 0;
12         runLength[0] = 1;
13         for(x=0; x<width; x++) {
14                 if(x > 0 && y > 0) {
15                         b22 = p[0] & p[-1] & p[-width] &
                               p[-width-1];
16                         w22 = p[0] | p[-1] | p[-width] |
                               p[-width-1];
17                         if((b22 | (w22 ^ 1))&1) {
18                                 demerit += N2;
19                         }
20                 }
21                 if(x == 0 && (p[0] & 1)) {
22                         runLength[0] = -1;
```

```
23                              head = 1;
24                              runLength[head] = 1;
25                      } else if(x > 0) {
26                              if((p[0] ^ p[-1]) & 1) {
27                                      head++;
28                                      runLength[head] = 1;
29                              } else {
30                                      runLength[head]++;
31                              }
32                      }
33                      p++;
34              }
35              demerit += Mask_calcN1N3(head+1, runLength);
36      }
37
38      for(x=0; x<width; x++) {
39              head = 0;
40              runLength[0] = 1;
41              p = frame + x;
42              for(y=0; y<width; y++) {
43                      if(y == 0 && (p[0] & 1)) {
44                              runLength[0] = -1;
45                              head = 1;
46                              runLength[head] = 1;
47                      } else if(y > 0) {
48                              if((p[0] ^ p[-width]) & 1) {
49                                      head++;
50                                      runLength[head] = 1;
51                              } else {
52                                      runLength[head]++;
53                              }
54                      }
55                      p+=width;
56              }
57              demerit += Mask_calcN1N3(head+1, runLength);
58      }
59
60      return demerit;
61      }
```

Nella prima porzione di codice si analizza la presenza di blocchi di moduli di uno stesso colore, per cui ammettendo di trovare un blocco m x n di medesimo colore, si andrà a aumentare il punteggio di penalizzazione del simbolo di un fattore pari a $(m-1)(n-1)N_2$, dove N_2 è la costante di demerito assegnata a questa caratteristica negativa; la variabile *demerit* al rigo 18 viene quindi incrementata

di N_2 per il giusto numero di volte. In corrispondenza delle linee 35 e 57 alla variabile demerit vengono invece addizionati valori proporzionali alle costanti di demerito N_1 e N_3, rispettivamente nei casi in cui ci siano molti moduli adiacenti dello stesso colore in una riga o colonna, oppure qualora i moduli acquisiscano per un caso sfortunato una configurazione simile a quella dei position detection pattern (ovvero il rapporto 1:1:3:1:1 tra moduli chiari e scuri). Quest'ultimo caso è molto grave per cui la costante N_3 ha un valore molto più alto delle altre costanti di penalizzazione.

- qrenc.c: Si tratta semplicemente del modulo che contiene la funzione main del programma, con annessa la gestione dei parametri ricevuti da linea di comando.

- qrencode.c: Rappresenta un modulo centrale per la codifica del simbolo, in quanto si occupa di riempire la struttura matriciale del simbolo (frame filling), distribuendo le codewords appartenenti alla sequenza finale precedentemente calcolata nelle giuste posizioni[7], in accordo alle specifiche.

```
typedef struct {
        int version;          // version of the symbol
        int width;            // width of the symbol
        unsigned char *data; // symbol data
} QRcode;
```

L'estratto soprastante appartiene al codice originale, in cui si vede come la struttura QRcode sia composta da tre campi, ovvero un intero per la versione del codice, un intero per l'ampiezza del simbolo in numero di moduli ed infine un puntatore ad un'area di memoria, che sebbene sembri rappresentare un array monodimensionale verrà invece gestito in termini di matrice. Ogni elemento di tale matrice corrisponde ad un modulo appartenente ad un function pattern

[7]Per dettagli ulteriori si veda la sezione 2.3.3, in relazione al paragrafo relativo al posizionamento delle codewords nell'area dati del simbolo.

oppure all'area dati che quindi porterà esattamente un bit di informazione. Nel contesto del nostro progetto di sviluppo di codici Quick Response a colori, si ha tuttavia la necessità di condensare più di un bit di informazione in ogni modulo: fermo restando che la maniera di spalmare i dati relativi alla sequenza finale nel simbolo viene ripresa anche nel nostro progetto, tuttavia, dal momento che si vuole fare in modo che ogni modulo rappresenti più bits, molte procedure vanno rielaborate per consentire un frame filling diverso, così come la definizione delle strutture coinvolte. Si propone nel seguito un estratto relativo alla struttura QRcode rielaborata per la produzione di codici Quick Response a maggiore capacità, a semplice titolo di esempio, non potendo illustrare qui i vari sorgenti con completezza. Innanzitutto compare un nuovo campo che contiene il valore relativo al numero di bits espressi da ogni modulo del simbolo; inoltre il puntatore *data* viene sostituito con il puntatore a puntatore di caratteri *bitmatrices_data*, che conterrà i dati spalmati su un numero di matrici pari proprio al valore di bits_per_module. Si aggiunge un'ulteriore matrice di luminosità, dipendente dalle altre matrici date da *bitmatrices_data*, che va a risolvere le problematiche relative al mascheramento di codici Quick Response a colori, consentendo di scegliere la maschera migliore per il simbolo in questione.

```
typedef struct {
        int bits_per_module;     // number of bits per module
        int version;             // version of the symbol
        int width;               // width of the symbol
        unsigned char *luminance_data;  // luminance symbol data
        unsigned char ** bitmatrices_data;      // symbol data
            splitted in bits_per_module bit matrices

} QRcode;
```

- qrinput.c: Si occupa di gestire problematiche relative all'input, come ad esempio convertire i dati di input in un bitstream oppure come capire quale versione del

codice sia la minima necessaria per la codifica di quegli specifici dati in ingresso.

- qrspec.c: Si tratta del modulo che racchiude tutte quelle informazioni relative alle specifiche che sono state pubblicate in varie tabelle sul documento relativo allo standard. Quindi, si trovano informazioni circa il numero di blocchi di Reed-Solomon che compongono ogni diversa versione dei codici Quick Response, il numero di bits di resto, il numero di moduli complessivi, la massima quantità di codewords rappresentabile in un dato tipo di simbolo, e così via. In sezione 3.3.1, si era definito come estendere le tabelle dello standard per rendere operativa la nuova simbologia prodotta; il sorgente è stato quindi rielaborato per soddisfare le nuove specifiche.

```
static int QRspec_getDataLength(int version, QRecLevel level)
{
        return qrspecCapacity[version].words -
            qrspecCapacity[version].ec[level];
}
```

A titolo di esempio, nell'estratto soprastante è riportata la funzione che recupera dalle tabelle il numero di codewords di dati (escluse quindi le codewords di correzione d'errore) per un dato simbolo, così come veniva usata dal sorgente originale per i Quick Response standard; nel nostro caso invece utilizzeremo la porzione di codice dell'estratto sottostante che in questo semplice caso effettua il semplice prodotto del valore originale recuperato e del numero di bits che ogni modulo rappresenta, in accordo alle specifiche proposte per la nuova simbologia.

```
int __modded_QRspec_getDataLength(int version, QRecLevel level,
    int bits_per_module)
{
        return bits_per_module * QRspec_getDataLength(version,
            level);
}
```

- rscode.c: Modulo di supporto alla funzionalità di correzione d'errore. Troviamo quindi l'implementazione dell'algoritmo di Reed-Solomon a cui si è più volte accennato.

- split.c: Gestisce l'operazione di "split", la suddivisione della sequenza da elaborare in accordo ai diversi "Modes".

4.5 Realizzazione del Lettore

Rielaborando il codice del progetto opensource Zxing, come si è già accennato in sezione 4.1, è stato possibile produrre un lettore di codici Quick Response a colori che fosse in grado di identificare, elaborare e decodificare con successo simboli appartenenti alla nuova simbologia introdotta, tanto in ambito mobile quanto in ambiente desktop; grazie a questo progetto a sorgente aperto si è potuto sviluppare un lettore in tempi più brevi e con risultati probabilmente migliori di quanto possibile altrimenti.

Il codice del progetto è scritto in Java, cosa che già di base garantisce una certa portabilità, tuttavia ZXing cerca di centrare questo obiettivo in modo in particolare, anche considerando che come si è detto in sezione 4.3.1, la programmazione in ambiente Android tende a perdere questa caratteristica, divenendo un mondo a parte per l'assenza di supporti come le piattaforme Java Standard Edition e Java Micro Edition. Dal momento che Zxing riesce con uno sforzo addizionale a garantire sufficiente portabilità, l'intera rielaborazione che è stata fatta per consentire la lettura della nuova simbologia di codici cercherà di fare altrettanto, mantenendo questa importante caratteristica.

Un secondo importante vincolo sul lavoro svolto è stato quello di garantire massima separabilità tra il codice del progetto originale ed i moduli software introdotti per

gestire i codici Quick Response ad alta capacità: questa importante caratteristica consente di produrre versioni distinte del nuovo lettore a partire da diverse versioni del progetto ZXing di riferimento.

In fondo, il progetto ZXing va tuttora avanti, essendo ancora attivo e con grandi possibilità di miglioramento, dunque la situazione ottima che si cercava era esattamente questa: avere a disposizione un insieme di componenti software facilmente innestabili in una nuova versione del progetto Zxing originale, ogni qualvolta venga rilasciata un nuovo build. Nel paragrafo 4.5.1 vediamo quale struttura del lavoro ha potuto consentire tutto questo.

4.5.1 Struttura del Progetto

Al fine di consentire il raggiungimento di obiettivi importanti come portabilità e separabilità dei componenti software addizionali per la gestione della nuova simbologia, si è fatto un utilizzo pesante di ereditarietà delle classi e di overidding, ovvero della ridefinizione, nelle sottoclassi, di vari metodi ereditati dalle superclassi.

In questo senso il progetto elaborato mira ad estendere le classi del progetto originali, ridefinendo tramite overidding tutto quell'apparato di metodi che va rielaborato per consentire il raggiungimento dei nostri scopi di decodifica di simboli Quick Response a colori.

Il progetto in questione è di dimensioni abbastanza importanti rispetto a quelle proprie del generatore prodotto, la cui struttura era stata sinteticamente descritta in sezione 4.4.1; infatti il progetto consta di diversi packages per un totale complessivo di diverse decine di files sorgenti.

Il lavoro di rielaborazione è stato focalizzato da un lato sulla ridefinizione di quei metodi da riadattare per rendere operativa la decodifica dei nuovi simboli, dall'altro

sull'introduzione di moduli ex novo per consentire alle operazioni di image processing di tenere in considerazione anche le problematiche cromatiche di interesse; nel fare questo si sono avuti sforzi congiunti nella direzione di mantenere tali funzioni ancora valide per l'elaborazione di codici Quick Response standard e nei limiti del possibile nel renderle facilmente innestabili sulle eventuali future versioni del progetto ZXing di riferimento.

Per dare un'idea del lavoro che è stato richiesto per l'estensione del progetto, possiamo dire che il sottoinsieme dei sorgenti della libreria ZXing che è stato rielaborato, unitamente alle nuove classi introdotte per le nuove necessità sorte dall'uso dei colori, supera le venti unità.

Nel seguito si cercherà di fare una panoramica su quali componenti, in relazione alla struttura dei packages e quindi dei sorgenti, sia stata concentrata l'opera di rielaborazione, nonché quali nuove classi siano state introdotte; in questo modo si avrà maggiore chiarezza sul lavoro svolto.

Si tratteranno nell'ordine i packages *com.google.zxing*, *com.google.zxing.qrcode*, *com.google.zxing.qrcode.detector*, *com.google.zxing.qrcode.decoder* e *com.google.zxing.qrcode.fullcolor*, tutti relativi al "core" del progetto ZXing, di cui soltanto l'ultimo è aggiunto ex novo.

Infine si descriveranno brevemente quei packages dipendenti dall'architettura, relativi, in primis, alla piattaforma Android ed in secondo luogo alla Java Standard Edition per i comuni pc; rispettivamente *com.google.zxing.client.android* per Android e *com.google.zxing.client.j2se* per J2SE.

Si noti che ogni qualvolta una classe che viene aggiunta al progetto erediterà da una classe presente tra i sorgenti originali, questa verrà contrassegnata con la nomenclatura *Mod-*{*nome classe originale*}.

4.5.2 Packages appartenenti al "Core": Indipendenti dall'Architettura

Iniziamo col ragionare relativamente a tutti quei componenti software che sono completamente indipendenti dall'architettura, essendo focalizzati a realizzare il funzionamento del nucleo applicativo. Tali componenti sono mantenuti completamente separabili da quelle porzioni di codice che invece fanno assunzioni sul dispositivo effettivo che ne lancerà l'esecuzione; al fine di garantire questa caratteristica, come ci si può aspettare, nessuna funzione che faccia parte del core può invocare metodi forniti esclusivamente dalla piattaforma Android, da Java Standard Edition o Java Micro Edition, potendo quindi utilizzare soltanto quel sottoinsieme di funzioni comuni.

- Per quanto riguarda il package *com.google.zxing*, che è il più generico tra quelli trattati essendo in realtà in comune con i componenti software legati alla decodifica di altre simbologie di codici[8], si è prodotto quanto segue:

  - La classe ModBinaryBitmap, che eredita da BinaryBitmap del progetto originale, ottiene nuovi attributi relativi all'immagine sorgente, affinché si possa supportare un image processing che tenga in considerazione le problematiche cromatiche nel caso in cui si riconoscano codici QR nel fotogramma percepito dal lettore ottico (scanner o fotocamera del dispositivo). La classe BinaryBitmap originale teneva traccia della sorgente di luminosità ma perdeva le informazioni cromatiche, fatto bloccante per il nostro lavoro.
  - La classe ModMultiformatReader, che eredita da MultiformatReader, viene rielaborata per abilitare il reader relativo alla nuova simbologia. Questo

[8]Si ricorda che Zxing supporta la decodifica di diverse varietà di codici, come EAN-8 ed EAN-13 per quanto riguarda i simboli monodimensionali oppure QR e PDF 417 per quanto riguarda i codici bidimensionali.

componente software ha infatti lo scopo di specificare quali tipi di lettore rendere operativi e quali disabilitare; si può ad esempio fare in modo che l'applicativo prodotto abiliti o disabiliti a runtime la rilevazione dei codici monodimensionali o dei Quick Response.

- Per quanto riguarda il package *com.google.zxing.qrcode*, questo è ancora un package abbastanza generico, tuttavia è relativo soltanto alla porzione del progetto riservata ai Quick Response. Qui troviamo, come era facile aspettarsi, la classe principale relativa al lettore per questo tipo di simboli.

  – Viene introdotta la classe ModQRCodeReader, che eredita da QRCodeReader, ovvero la classe principale[9] relativa al "core" di decodifica dei Quick Response. Qui va completamente ridefinito il metodo *decode()* in maniera da poter lanciare i diversi moduli introdotti o rielaborati che costituiscono il nostro lavoro, per i quali in questa sezione si sta cercando di tracciare quantomeno un quadro di riferimento. Come linea guida ad alto livello, in primis si deve far lavorare il metodo *detect()*, che è proprio di un componente Detector che è stato riadattato in questo lavoro per la rilevazione di simboli Quick Response a colori nei frames relativi agli scatti della fotocamera; in secondo luogo si deve lanciare il metodo *decode()* di un componente Decoder che è stato rielaborato per una decodifica che sia in accordo con quelle specifiche che sono state proposte nel capitolo 3.

- Descriviamo a questo punto, seguendo la scaletta proposta, che cerca di seguire in qualche modo il susseguirsi di interazioni tra i moduli software, prima il pack-

[9] Si parla di classe principale soltanto in relazione al "core", poiché vi è poi quella parte di codice dipendente dalla piattaforma, come ad esempio relativa ad Android o per i comuni pc, che avrà una propria classe principale, la quale si farà carico di interagire proprio con quella del core.

age relativo al Detector, ovvero *com.google.zxing.qrcode.detector*, per passare poi a quello relativo al Decoder. Il Detector, come è stato già accennato, focalizza il suo lavoro alla rilevazione della presenza di un certo tipo di simbolo di interesse all'interno di una data immagine, che sarà una scansione proveniente da uno scanner o dalla fotocamera dello smartphones. Si occupa quindi di fare quel lavoro a monte di tutta la questione, per cui, data un'immagine digitale, in cui non si sa cosa vi sia rappresentato, si cerca di identificare la presenza del simbolo, ammesso che la scansione lo contenga. Nel fare questo non può fare assunzioni circa l'effettiva dimensione del codice nell'immagine, circa la sua orientazione o la luminosità delle sue componenti, dovendo quindi svolgere un lavoro molto delicato di processamento dell'immagine. Diamo una sintetica descrizione di quanto prodotto in merito.

- Si aggiunge la classe ModDetector, che eredita dal Detector originale e che punta ad abilitare la rilevazione di simboli Quick Response a colori mediante l'invocazione di un algoritmo campionatore dell'immagine (sampler) riprogettato ad hoc, nonché di un modulo valutatore delle componenti cromatiche delle legende replicate (qualora siano presenti), introdotto ex novo.
- Come è stato accennato va riprogettato il campionamento dell'immagine: il compito spetta alla classe ModDefaultGridSampler, la quale ora deve, in aggiunta, sia effettuare tale operazione nei punti rappresentativi delle legende replicate, sia convertire diversamente il valore analogico[10] del modulo

[10]L'immagine è digitale, non analogica, tuttavia se pensiamo che i possibili valori per ogni pixel possono arrivare anche a 16 milioni, si può quasi assumere che possano prendere qualunque gradazione reale. L'abuso di notazione serve solo a sottolineare come il valore cromatico di un pixel che può assumere milioni di diverse gradazioni va elaborato e convertito in digitale, in una sequenza di 1,2,3

campionato.

  - ModDetectorResult è una classe aggiuntiva da rielaborare soltanto come conseguenza per l'aver riprogettato il Detector; una volta modificato quel componente va ovviamente modificata anche la classe che ne condensa il risultato.

  - La classe PaletteFinder è invece aggiunta ex novo (e dunque non poniamo il prefisso *Mod-*); il suo scopo è di elaborare i punti dell'immagine relativi alle legende replicate, campionati dall'algoritmo di sampling, al fine di produrre dei valori cromatici di riferimento per la futura conversione in digitale dei moduli. Qualora si rilevi l'assenza del pattern dato dalle legende la sua funzione sarà invece di riportare l'elaborazione nella direzione della decodifica dei Quick Response standard (che si ricorda, non posseggono tali legende).

- Per quanto riguarda i moduli software relativi al Decoder, questi sono concentrati nel package corrispondente, vale a dire *com.google.zxing.qrcode.detector*, e per almeno cinque di loro è necessario una profondo riadattamento, in virtù della necessità di andare incontro alle specifiche riprogettate nel capitolo 3.

  Nota editoriale (2026): In questo punto il testo del 2010 riporta per evidente refuso il package `com.google.zxing.qrcode.detector`; il contesto e le classi elencate subito sotto identificano invece `com.google.zxing.qrcode.decoder`, denominazione usata nella traduzione inglese.

  - La classe ModVersion, che eredita da Version, è probabilmente quella che più di tutti risponde alla necessità di estensione delle tabelle dello standard Quick Response, che erano state discusse in sezione 3.3.1. Si devono ad esempio parametrizzare qui sia il numero di blocchi di Reed-Solomon

o anche 4 bits se si pongono soglie di riconoscimento per 16 colori.

costitutivi del simbolo, come anche il numero di codewords codificabili in esso.

  - ModMode completa l'opera iniziata da ModVersion: volendo rispettare i vincoli imposti dalla nuove specifiche proposte, relativamente all'utilizzo dei vari modes, si devono infatti riadattare i valori relativi alle dimensioni del character count indicator, al fine di evitare che tale contatore non sia in grado di rappresentare l'effettivo numero di caratteri codificati nel simbolo a causa di un quantitativo di bits, a lui allocati, eccessivamente ridotto.
  - ModDecodedBitStreamParser e ModBitMatrixParser vengono introdotte al fine di consentire che i parsing della varie matrici di bits utilizzino, per le loro computazioni, informazioni di versione e formattazione provenienti da una matrice di bits terza, ovvero quella relativa ai valori di luminosità.
  - La classe ModDecoder, che eredita da Decoder, è la classe fondamentale di questo package, che si occupa di lanciare i vari algoritmi necessari per la decodifica, incluso quello relativo alla correzione d'errore. Tuttavia, al fine di avere una sequenza finale da fornire in input all'algoritmo di Reed-Solomon, si deve pensare di interlacciare i dati parsati da ogni singola matrice di bits in maniera opportuna; passo non necessario ovviamente per la decodifica dei codici Quick Response standard.

- Per quanto riguarda il package *com.google.zxing.qrcode.fullcolor*, questi è del tutto introdotto ex novo. L'obiettivo di tale package è quello di raggruppare tutti quei componenti software strettamente correlati alla gestione delle problematiche cromatiche dell'immagine scansionata; Descriviamo ancora una volta in maniera sintetica l'obiettivo di ciascun componente introdotto.

- Si introduce un ColorBinarizer, un binarizer riprogettato ad hoc per la conversione dei valori cromatici campionati dell'immagine nelle sequenze di bits che vengono estratte. Al fine di ottenere questo risultato va utilizzata una metrica di distanza cromatica, la cui necessità era stata già descritta in sezione 3.3.4.
- ColoredPixel, RgbPixel e YCbCrPixel sono classi utilizzate per la rappresentazione dei valori dei pixels nei due diversi formati, consentendo anche possibilità di conversione dall'uno all'altro.
- FullColorImageHandler, insieme alle varie sottoclassi (YCbCrColorImageHandler, YCbCr420SpImageHandler, RgbColorImageHandler), adempie alla gestione dei dati relativi all'immagine scansionata, fornendo metodi di manipolazione della stessa.

4.5.3 Packages Dipendenti dalla Piattaforma

Descriviamo ora brevemente a quali packages dipendenti dalla piattaforma è stato necessario apporre delle modifiche per abilitare l'esecuzione di quei componenti software, introdotti nel nucleo applicativo, che sono stati descritti precedentemente.

- Per quanto riguarda il sistema operativo Android, la cosa che va sicuramente abilitata è la possibilità di mantenere l'informazione cromatica dell'immagine digitale ottenuta a seguito dello scatto della fotocamera. Nel progetto originale, dato che parte dell'array di dati grezzi in formato YCbCr non era necessario, si catturavano i valori di luminosità dell'immagine, gettando invece le componenti cromatiche (peraltro sottocampionate) relative al rosso e al blu. Il package di riferimento relativo alla porzione di codice dell'applicazione volta alla piattaforma Android è dato da *com.google.zxing.client.android*; la classe che va princi-

palmente riadattata è DecodeThread, estesa in ModDecodeThread secondo la solita consuetudine. La rielaborazione è necessaria per gestire diversamente il thread che produce di volta in volta nuovi frames, a seguito degli scatti della fotocamera, al fine di permettere il continuo "rifornimento" di immagini digitali a colori nei confronti dei componenti del nucleo applicativo.

- Relativamente alla piattaforma Java Standard Edition, la gestione risulta semplificata in quanto l'immagine scansionata non avrà l'informazione cromatica sottocampionata ed inoltre ci si potrà aspettare la classica rappresentazione cromatica dei pixels costitutivi nel formato rgb. Nel package corrispondente, *com.google.zxing.client.j2se*, è necessario un semplice riadattamento della classe GUIRunner per abilitare l'esecuzione del nuovo decoder. In futuro si potrebbero aggiungere filtri all'immagine scansionata per il miglioramento delle prestazioni, poiché l'ambiente desktop può mettere a disposizione processori di prestazioni molto elevate rispetto a quanto l'ambiente mobile possa offrire.

4.5.4 Caratteristiche di Retro-Compatibilità

L'applicazione di decodifica dei simboli prodotta, continua a rilevare e decodificare con successo i codici Quick Response Standard: in questo senso, vi è un lavoro iniziale comune alla decodifica dei QR standard e dei QR a colori, che vogliono estendere i primi mantenendo però retro-compatibilità. Lo sforzo computazionale in comune, è almeno di trovare i position detection patterns, di capire le posizioni in cui campionare i moduli, di leggere informazioni di versioni e formattazione. Se così non fosse non potremmo dire che i QR standard possono essere considerati sottoinsieme dei QR a colori ad alta capacità. Dopo il processamento comune, l'algoritmo di decodifica

cerca di capire se siano o meno presenti le legende[11]: se non lo sono si può procedere con le solite computazioni, mentre, qualora le legende vengano rilevate, si possono lanciare quei moduli software introdotti per gestire l'estensione di processamento. Questo sistema garantisce di aggiungere sforzo computazionale solo se necessario, o solo un piccolo overhead qualora non lo sia.

Osservazione Immaginiamo di trovarci di fronte un codice Quick Response standard e di volerlo gestire con il lettore esteso per la decodifica dei QR a colori ad alta capacità; se confrontassimo le computazioni svolte da tale lettore con quelle eseguite dal decoder dei QR standard, potremmo dire che svolgeremmo in più solo la ricerca delle legende (la quale ovviamente sarebbe inutile e fallirebbe in questo caso di tentativo di lettura di un QR standard). L'overhead introdotto dalla rilevazione delle legende pesa comunque ben poco, se si pensa che queste possono essere replicate su una riga giustapposta al codice, e che, gestire una sola riga aggiuntiva in un simbolo QR che può avere dalle 21 alle 177 righe, non può rappresentare uno sforzo aggiuntivo eccessivo. Questo discorso vale a prescindere dalla ubicazione delle legende, fermo restando che si potrebbe pensare in futuro di riposizionare tale pattern diversamente. Nel capitolo 5 si punterà a valutare misurazioni quantitative dell'overhead introdotto, al fine di provare quanto è stato per ora solo ipotizzato.

4.5.5 Sintesi di Funzionamento dei Moduli Software Introdotti

Analogamente al ragionamento fatto precedentemente in sezione 4.5.4, ragioniamo sul secondo scenario, quello relativo al dover decodificare un codice appartenente alla

[11] Il nuovo pattern introdotto al fine di venire incontro ad esigenze di generalizzazione degli schemi di colori utilizzabili, nonché, fatto più importante, voler rispondere alla necessità di correzione delle distorsioni cromatiche.

nuova simbologia. A differenza di quanto avviene in corrispondenza del caso base relativo ai QR standard, qualora il lettore si trovasse a dover elaborare una scansione in cui sia presente un simbolo a colori, dovrà lanciare l'esecuzione di appositi moduli software aggiuntivi che sono stati introdotti nella rielaborazione del progetto ZXing di partenza.

Poniamoci a livello di quell'entità software a cui vogliamo dare l'appellativo di modulo "Detector"[12]: immaginiamo che siano stati rilevati i position detection patterns, valutate le dimensioni unitarie dei moduli[13], la dimensione dell'intero simbolo in numero di celle e di conseguenza anche la (probabile) versione del simbolo Quick Response che si ha di fronte; assumiamo che siano stati riconosciuti gli alignment patterns e verificato che le legende siano effettivamente presenti in questo ipotetico codice QR di esempio. A questo punto, relativamente allo scenario ipotizzato, il risultato finale del processo di campionamento non potrà più essere semplicemente una matrice di bits che assegna in ogni posizione (i, j) il valore 0 o 1 a seconda, rispettivamente, dell'alta o bassa luminosità del modulo campionato; ogni pixel, che venga scelto come rappresentativo di un certo modulo, dovrebbe infatti essere rapportato ai colori campionati nella legenda, e, in base alla metrica di distanza cromatica scelta, si dovrebbe stabilire quale colore associargli e quindi quali bits (ora più d'uno) fargli corrispondere.

I dati estratti tramite campionamento cromatico dei moduli sono tuttavia, anche arrivati a questo punto, ancora mascherati. Gli algoritmi per l'operazione standard di unmasking prendono tipicamente in input una matrice di bits ed applicano l'oper-

[12]Definiamo come "detector", in questo contesto, l'insieme delle funzioni predisposte (almeno) all'identificazione ed alla rilevazione della presenza di simboli nelle immagini, nonché l'analisi di orientazione dei codici nelle stesse. Nel progetto ZXing il codice di queste procedure si concentra nel package com.google.zxing.qrcode.detector

[13]Esprimendo tali quantità come numero di pixels per modulo relativamente alla specifica scansione su cui si sta lavorando.

azione di xor secondo lo specifico mask patter che era stato usato in fase di codifica, invertendo il processo ed ottenendo i dati originali. In riferimento a questa considerazione, si è pensato che la soluzione più lineare, che ottimizza anche lo spazio allocato alle strutture dati di riferimento, è di memorizzare questi dati usando n matrici di bits, con n pari al numero di bits rappresentati da ogni modulo, in aggiunta ad una matrice di bits rappresentativa dei valori di luminosità. Su questa e su ognuna delle n matrici possiamo singolarmente effettuare l'operazione di unmasking ed estrarre il flusso di bits come da standard, come si fa quindi di consueto quando si lavora su una sola matrice rappresentativa della luminosità. Ottenuti i vari flussi di dati non mascherati, viene richiesto un passo addizionale: al fine di ottenere un'unica sequenza di bits si devono interlacciare nella giusta maniera gli n flussi ottenuti da ciascuna delle n matrici, così da avere una sequenza pronta per la successiva fase di correzione degli errori. L'ultima matrice di riferimento da trattare, ovvero quella rappresentativa di luminosità, la utilizziamo per ottenere le informazioni di versione e formattazione come usuale.

Con un meccanismo del genere, di cui è stata proposta una sintesi che, seppur non potendo perseguire un obiettivo di completezza vista la complessità dei dettagli interni, si è ridefinita la maniera di fare unmasking qualora si usino più colori in un codice Quick Response. Si ricorda che affinché tutto ciò sia efficace, il generatore, nell'atto di scegliere la miglior maschera per la generazione del simbolo, si deve basare sulla matrice di luminosità del codice QR a colori, facendo corrispondere ad ogni modulo colorato ancora una volta 0 se è relativo ad un colore chiaro, 1 se scuro, similmente ai moduli b/n trattati dallo standard. Inoltre qualora ad un colore luminoso corrisponda una certa sequenza di bits, alla sequenza di bits ottenuta invertendo ogni singolo bits della prima sequenza deve corrispondere un colore scuro, e viceversa. Se

queste condizioni non valgono il processo citato perde di valore e di efficacia.

Finalmente, dopo questa prima elaborazione, si può dare in input al modulo software a cui diamo l'appellativo di "Decoder"[14] il flusso di bits interlacciato prodotto al passo precedente; questi dovrà interpretare la sequenza di bytes come suddivisa in un certo numero di blocchi di reed-solomon, cercando di correggere le codewords errate, prima di poter andare avanti con il processo di decodifica. Affinché quest'ultima vada a buon fine, si ricordi di parametrizzare con il valore dei bits_per_module i dati delle tabelle relativi al modulo Version, nonché di modificare il modulo relativo ai Modes, al fine di correggere i valori relativi al character count indicator allo stesso modo di quanto fatto durante il lavoro di rielaborazione del generatore.

[14]Con il termine "decoder" si vuole intendere tutto l'insieme di componenti software del progetto che hanno lo scopo di decodificare i dati estratti, escludendo quindi tutte quelle funzioni di processamento dell'immagine necessarie a recuperare tali dati. Nel progetto Zxing queste procedure si concentrano nel package com.google.zxing.qrcode.decoder

Capitolo 5

Sperimentazione ed Analisi dei Risultati

In questo capitolo si vogliono mettere a fuoco le prestazioni del lettore dei codici Quick Response a Colori prodotto, a seguito dei processi di stampa [7], su carta comune con diversi gradi di qualità, e di scansione, tramite un normale scanner per l'ambiente desktop e tramite la fotocamera di alcuni smartphones per l'ambiente mobile.

Dal momento che sono stati realizzati dei prototipi sia per un generatore che per un lettore di codici Quick Response a colori, di cui si è cercato di fornire una descrizione nel capitolo 4, si vuole fare il punto della situazione sui risultati ottenuti. A livello di componenti hardware, per quanto riguarda l'ambiente desktop, si è avuto a disposizione uno scanner/stampante multifunzione Epson Dx6000 che relativamente alla stampa si basa su cartucce di inchiostro, mentre per quanto riguarda l'ambito mobile, sono stati fatti dei test relativamente all'uso di smartphones, quali htc tattoo e htc magic, come lettori ottici della nuova simbologia di codici, previa installazione del nostro applicativo. Entrambi gli smartphones citati sono gestiti da un software di base costituito dal sistema operativo Android O.S., nello specifico la versione 1.6, anche detta Android Donut. Nonostante siano dispositivi mobili con caratteristiche simili per molti versi, le prestazioni registrate nei due diversi casi sono state ragionevolmente diverse in virtù del fatto che mentre htc magic ha una fotocamera provvista di

autofocus, htc tatto ne è completamente sprovvisto.

Se è vero che l'assenza di autofocus può diventare poco determinante in alcuni tipi di fotografie di uso comune, non è sicuramente così in relazione agli scatti riguardanti i codici bidimensionale o in generale qualunque tipo di codice, anche a barre; infatti qualora si scenda sotto una certa soglia per quanto riguarda le dimensioni di stampa dei codici, tali simboli diventano de facto illeggibili per tutti quei dispositivi le cui fotocamere non siano provviste di autofocus.

5.1 Esempi di Funzionamento

Finora si è descritto tutto il procedimento intrapreso per arrivare a produrre dei codici Quick Response a colori che consentano un incremento della densità di informazione, tuttavia non si è ancora mostrato un solo esempio di codici appartenenti alla nuova simbologia. Dunque, si vogliono far vedere alcune istanze di simboli generati mediante quel prototipo di codificatore che abbiamo trattato in sezione 4.4.

Figura 5.1: Esempio di codice di tipo 2H che utilizza uno schema a 4 colori (2 bits/modulo) per la rappresentazione ottica dei dati.

Nota editoriale (2026): La didascalia è stata modificata per l'edizione digitale del 2026 eliminando il riferimento diretto a "Quick Response". L'immagine è invariata e documenta un risultato prodotto dal prototipo dell'autore nel 2010.

Figura 5.2: Esempio di codice di tipo 6H a 4 colori (2 bits/modulo).

Nota editoriale (2026): La didascalia è stata modificata per l'edizione digitale del 2026 eliminando il riferimento diretto a "Quick Response". L'immagine è invariata e documenta un risultato prodotto dal prototipo dell'autore nel 2010.

Relativamente ai simboli soprastanti, si mostrano nel seguito gli screens relativi al successo dell'operazione di lettura avvenuta tramite il dispositivo htc tattoo, a partire da codici stampati su carta comune. Ragionando su questo dispositivo, la risoluzione è di soli 320x240 pixels, gli scatti non possono usufruire di autofocus e vi è quel sottocampionamento delle sfumature cromatiche che era stato descritto in sezione 4.3.3. Si noti quindi che si è riusciti a recuperare le informazioni contenute nei simboli di figura 5.1 e 5.2 a partire da un'immagine digitale che pesa soli 115200 bytes (si calcola a partire dal prodotto 320x240, i cui valori essendo relativi alla risoluzione ci danno il numero di pixels complessivi; dopodiché si moltiplica tale prodotto per 3/2 dal momento che mediamente ogni pixel viene rappresentato da un byte e mezzo, ovvero 12 bits).

Figura 5.3: Screen relativo alla lettura del simbolo di figura 5.1 su htc tattoo.

Nota editoriale (2026): Questo screenshot storico è invariato e documenta il prototipo e la sperimentazione dell'autore del 2010.

Figura 5.4: Screen relativo al funzionamento dell'applicativo su htc tattoo per il simbolo in figura 5.2.

Nota editoriale (2026): Questo screenshot storico è invariato e documenta il prototipo e la sperimentazione dell'autore del 2010.

Si è detto che l'autofocus è importante per la lettura di codici di dimensioni ridotte, dunque mostriamo nel seguito uno screen di una decodifica ottenuta da htc magic che ne è provvisto. Si tratta di un codice di dimensione minore di un pollice quadrato, stampato su carta comune che, decodificato, ha portato quasi 400 bytes di contenuto. La risoluzione dello scatto è in questo caso superiore a quella vista precedentemente, attestandosi a 320x480 pixels.

Figura 5.5: Screen relativo alla lettura di un simbolo di versione 8L a 4 colori su htc magic. Si noti come la dimensione sia inferiore al pollice quadrato.

Nota editoriale (2026): Questo screenshot storico è invariato e documenta il prototipo e la sperimentazione dell'autore del 2010.

Gli esempi di screens che sono stati mostrati sono relativi all'esecuzione del prototipo di lettore su Android O.S., tramite utilizzo dell'hardware ottico degli smartphones; le scansioni ottenute tramite un comune scanner sono risultate essere decisamente migliori di quanto prodotto dagli scatti della fotocamera in modalità Camera Preview, anche per via dell'assenza del sottocampionamento cromatico,; si è quindi potuto, da un lato, leggere codici a 4 colori a maggiore densità, dall'altro, riuscire

nella decodifica di alcuni simboli a 16 colori, relativamente ai quali però rimane del lavoro futuro da fare, al fine di incrementarne affidabilità.

Figura 5.6: Screen relativo alla scansione di un simbolo a 16 colori in ambiente desktop, ottenuta tramite scanner Epson Dx6000, che è stata decodificata con successo.

Nota editoriale (2026): Questo screenshot storico è invariato e documenta il prototipo e la sperimentazione dell'autore del 2010.

5.2 Analisi dell'Overhead Introdotto

Sono stati eseguiti alcuni test prestazionali sul prototipo prodotto mirati a capire quanto è costato, in termini di costi computazionali aggiuntivi, l'aver introdotto i colori nei codici Quick Response, con associate tutte le elaborazioni addizionali che la loro gestione comporta.

Un confronto quantitativo è facilmente ottenibile se si pensa che si potrebbe usare una stessa macchina per lavorare, in primis, con il lettore prodotto dal progetto ZXing originale, ed in secondo luogo, con il lettore esteso per la decodifica di Quick Response a colori; dal momento che quest'ultimo applicativo è basato su ZXing e ne possiede in comune varie porzioni di codice, il confronto può senz'altro avere senso e generare dei risultati che siano significativi.

Al fine di rendere valido un esperimento di questo tipo si sono presi in considerazione decine di simboli diversi, per ognuno dei quali sono stati effettuati 20 runs di decodifica al fine di calcolare una media dei tempi di esecuzione che possa essere presa come riferimento attendibile relativamente al tempo di lettura dello specifico codice. I tempi complessivi sperimentati sono ovviamente dipendenti dall'hardware che è stata utilizzato, dunque hanno senso solo se sono relativi ad esecuzioni avvenute sulla stessa macchina, nonché, per quanto possibile, calcolati in prossimità temporale gli uni dagli altri, al fine di smussare diversità prodotte da differenti condizioni di carico della macchina.

Tutti i risultati di test che seguono sono relativi ad esecuzioni in ambiente desktop su una macchina caratterizzata da un processore intel dual core 1.73 GHz, 3 Gb di ram, un software di base dato dal sistema operativo Slackware Linux 13.0 ed una condizione di carico leggera; i runs sono eseguiti di seguito, per le esigenze di prossimità temporale, mediante un'apposita applicazione di test che lancia continue esecuzioni calcolando le medie dei tempi ottenuti. Il tempo di ogni iterazione di decodifica, valutato in millisecondi, è definito come il tempo intercorso dal momento in cui il programma termina di caricare in ram l'immagine relativa alla scansione, subito prima di iniziare ad elaborarla, fino all'istante in cui il messaggio codificato è finalmente recuperato.

Seguono i dati ottenuti dal confronto delle esecuzioni dell'applicazione originale realizzata dal progetto ZXing all'opera su codici Quick Response standard, rispetto ad esecuzioni del prototipo di lettore prodotto in riferimento alla decodifica di codici Quick Response a colori. Si indicheranno con la dicitura "QR" i codici standard.

Nota editoriale (2026): Il seguito del periodo del 2010 impiegava una denominazione provvisoria, omessa in questa edizione. Nel confronto, "Simboli a 4 colori" e "Simboli a 16 colori" indicano i casi della nuova simbologia, denominata HCC2D nelle pubblicazioni successive alla tesi [4]. Il significato tecnico resta invariato.

Versione	QR	Simboli a 4 colori	Simboli a 16 colori
1L	122	132 (7.57%)	135 (9.62%)
1M	123	136 (9.55%)	138 (10.87%)
1Q	129	133 (3.01%)	135 (4.45%)
1H	131	135 (2.97%)	136 (3.68%)
5L	143	171 (16.38%)	206 (30.59%)
5M	151	174 (13.21%)	208 (27.40%)
5Q	163	181 (9.95%)	208 (21.63%)
5H	161	182 (11.54%)	209 (22.97%)
10L	176	209 (15.79%)	246 (28.45%)
10M	188	208 (9.61%)	249 (24.49%)
10Q	189	210 (10.0%)	273 (30.77%)
10H	190	212 (10.37%)	282 (32.62%)
20L	240	349 (31.23%)	387 (37.98%)
20M	253	334 (24.25%)	398 (36.43%)
20Q	250	337 (25.81%)	389 (35.73%)
20H	257	323 (20.43%)	395 (34.93%)
30L	333	447 (25.50%)	474 (29.75%)
30M	350	430 (18.60%)	460 (23.91%)
30Q	347	437 (20.59%)	478 (27.40%)
30H	338	416 (18.75%)	506 (33.20%)
40L	430	483 (10.97%)	550 (21.81%)
40M	415	481 (13.72%)	540 (23.15%)
40Q	420	500 (16.0%)	566 (25.79%)
40H	373	485 (23.09%)	552 (32.42%)

Figura 5.7: Risultati prestazionali relativi ai tempi medi di decodifica, espressi in millisecondi, per le diverse categorie di codici.

Nota editoriale (2026): I valori sperimentali sono invariati rispetto alla figura del 2010. La tabella è stata nuovamente composta per questa edizione e le denominazioni provvisorie delle colonne relative alla simbologia a colori sono state sostituite con descrizioni neutrali.

Nelle colonne relative ai codici Quick Response a colori, si può notare tra parentesi, a fianco del valore relativo alla media su 20 runs dei tempi di decodifica, anche la percentuale di tempo addizionale richiesto per la lettura del simbolo rispetto a quanto necessario per un codice Quick Response standard di pari versione e medesimo livello di correzione d'errore.

5.3 Affidabilità dei Codici

Descritto l'overhead introdotto a seguito delle nuove necessità di elaborazione, legate ad un processamento dell'immagine che tenga conto anche degli aspetti cromatici, si vuole ora ragionare sull'affidabilità relativa ai codici appartenenti alla nuova simbologia.

I test che sono stati portati avanti riguardano decine di codici a colori, che sono stati stampati su carta comune tramite stampante Epson Dx6000, ognuno avente la dimensione complessiva di un pollice quadrato; si sono inoltre tentate varie alternative cromatiche per gli schemi di colori costitutivi dei simboli.

Si è riscontrato che, a partire da uno scanner/stampante comune come può essere Epson Dx6000, è possibile leggere codici Quick Response a 4 colori, che superino la capacità di 1 kilobyte, che siano stampati in dimensioni pari ad un pollice quadrato e che siano scansionati a 600 dpi, con il solo livello L, al 7%, di correzione d'errore. Per codici di capacità leggermente più ridotta del kylobyte, ancora una volta stampati in un pollice quadrato, si possono usare anche basse qualità di stampa (ad eccezione della modalità bozza), e risoluzioni di scansione di appena 300 dpi, riuscendo a decodificare con successo i simboli.

Capitolo 6

Conclusioni e Sviluppi Futuri

Il presente lavoro di tesi ha introdotto una nuova simbologia di codici, basati sui Quick Response (codici QR), in grado di avvalersi dell'informazione cromatica per incrementare la densità di informazione; tale tecnologia è stata dunque sperimentata in ambiente desktop e su dispositivi mobili, previa realizzazione di applicativi di generazione e decodifica dei nuovi codici.

A livello di risultati ottenuti, le specifiche proposte sono state rilevate valide ed i codici prodotti sperimentati leggibili; sono stati condotti alcuni tests, sulle piattaforme desktop e mobile, che hanno mostrato il funzionamento dei processi di generazione e lettura, tenendo in considerazione le problematiche di stampa e di scansione dei simboli. La spinta all'innovazione dei codici è arrivata dalla grande diffusione di quei dispositivi mobili in grado di agire come lettori ottici, facendo uso della fotocamera integrata e previa installazione dell'opportuna applicazione. Per questo motivo, anche in considerazione delle potenzialità dell'hardware ottico, attuale e soprattutto futuro, in dotazione sui moderni smartphones, si è ritenuto interessante cercare di sfruttare appieno le nuove possibilità offerte dalla tecnologia, tentando di adoperare al meglio quell'informazione cromatica che le fotocamere sono perfettamente in grado di catturare.

In questo senso, si è percorsa differentemente una strada che è stata intrapresa anche da Microsoft, impegnata nel tentativo di promulgare la diffusione del codice "High Capacity Color Barcode", tecnologia di simboli bidimensionali a colori, come lo sono anche quelli proposti in questo lavoro, tuttavia rappresentativa di una soluzione proprietaria, che per giunta non possiede caratteristiche così rilevanti da renderla superiore ai codici bidimensionali preesistenti. La soluzione ricercata va invece nella direzione dell'introduzione dei colori in simboli già performanti e propri di una tecnologia matura, come può essere quella dei Quick Response, di cui peraltro sono completamente pubblicate le specifiche nel documento dello standard ISO/IEC 18004.

Per la realizzazione di questo lavoro sono state affrontate sfide non banali relativamente alle problematiche di image processing e di computer vision, di programmazione per dispositivi mobili con Android O.S., di riutilizzo di codice proveniente da progetti opensource scarsamente documentati, oltre alle sfide concettuali di rielaborazione delle specifiche dei codici Quick Response, che rappresenta il passo base su cui si è costruito il resto del lavoro.

Nonostante le prestazioni finali del prototipo siano soddisfacenti, si intravedono potenzialità legate a possibili sviluppi futuri. In primo luogo si consideri l'importanza di definire e sperimentare nuove metriche di distanza cromatica, al fine di trovare quelle che siano maggiormente in grado di valutare similarità e differenze tra i colori percepiti dal lettore ottico; ciò potrebbe consentire di abbassare la percentuale di errore nel riconoscimento cromatico di ciascun modulo, incrementando l'affidabilità dell'intero codice. Si potrebbero inoltre considerare eventuali ottimizzazioni relative al processamento dell'immagine, per una migliore identificazione del simbolo ed un campionamento più accurato dei moduli che lo compongono, oppure si potrebbe ragionare sui benefici che potrebbe portare l'applicazione di determinati filtri sull'im-

magine digitale. Un esempio è dato dal cosiddetto filtro di gauss, passa basso, che consente la sfocatura dell'immagine, garantendo così l'assenza di pixels erronei caratterizzati da colori completamente diversi da quelli ubicati nelle immediate vicinanze; poiché la gamma dei possibili filtri è molto ampia, sviluppi futuri possono andare nella direzione di analizzare quale possa essere il più indicato per migliorare la leggibilità dei simboli, valutando eventualmente anche l'utilizzo di più filtri in cascata, qualora il dispositivo abbia una sufficiente potenza computazionale. Ulteriori spunti di ricerca potrebbero riguardare attività volte a perfezionare l'ubicazione delle legende cromatiche che vengono nella versione attuale replicate lungo i lati del codice. In questo senso, si potrebbe pianificare di assegnare una differente superficie del codice per tali legende, qualora si verificasse che esistono posizioni migliori, capaci di garantire un campionamento cromatico di qualità superiore. Una prima idea consiste ad esempio nel replicare le legende quantomeno su due lati adiacenti, al fine di supportare la correzione di possibili disallineamenti tanto sull'asse orizzontale, quanto su quello verticale. La stessa scelta dei colori, alla base dello schema cromatico utilizzato come legenda, potrebbe essere analizzata accuratamente, in quanto se è vero che si vuole lasciare libero arbitrio al generatore su tale scelta, è anche vero che alcuni schemi di colore avranno prestazioni molto più elevate di altri e sarebbe interessante investigare quale soluzione si rivela infine più performante. Continuando a ragionare su possibili sviluppi futuri, si potrebbe lavorare sul mascheramento dei dati, al fine di incrementare, anche relativamente a questo ambito, l'affidabilità dei codici; la maniera di reinventare il Data Masking che è stata portata avanti consegue risultati interessanti, tuttavia un ulteriore elemento di ricerca potrebbe essere di valutare la bontà di possibili alternative in relazione a differenti scenari di luminosità, come ad esempio casi in cui si è in presenza di luce artificiale o di luce solare, situazioni di penombra, etc...

Si potrebbe ipotizzare infatti che la gestione delle diverse condizioni di luminosità sia più critica in presenza di codici a colori di quanto non avvenga per i classici codici bidimensionali in bianco e nero, con la conseguente necessità di un'analisi accurata nel merito. Infine, si potrebbe rivalutare la modalità in cui Android O.S. interagisce con la fotocamera, abbandonando la modalità Camera Preview, molto comoda per avere scatti consecutivi in maniera trasparente all'utente, ma allo stesso tempo povera di informazione cromatica, che, come è stato descritto nel corso della trattazione, punta ad un sottocampionamento di tali informazioni.

Elenco delle figure

1.1 Un esempio di Codice a Barre Ean-8. 9

1.2 Confronto tra un codice 1D (a barre) e un codice 2D (qui di tipo QR) 11

1.3 Confronto tra le principali famiglie di codici bidimensionali e le rispettive capacità massime. 13

1.4 Un esempio di codice Microsoft HCCB - Microsoft Tag 14

1.5 Struttura del codice PDF417 16

1.6 Esempio di codice Codablock-F 16

1.7 Struttura del codice Data Matrix 17

1.8 Codici Microsoft HCCB a 4 ed a 8 colori 22

3.1 Relazioni concettuali fra i parametri di capacità discussi nel testo del 2010. 35

3.2 Dipendenze concettuali fra versione, correzione d'errore, quantità di dati e capacità in ingresso. 38

3.3 Relazioni concettuali impiegate nell'estensione dei parametri di correzione d'errore. 40

3.4 Logica concettuale di estensione del Character Count Indicator. . . . 44

3.5 Schema creato dall'autore delle regioni funzionali di un simbolo Model 1. 49

3.6 Esempio di Data Masking in accordo allo standard Quick Response . 52

3.7 Esempio di Data Masking rielaborato per la nuova simbologia 53
3.8 Soluzioni alternative per l'organizzazione interna dei moduli colorati delle legende. Ai colori effettivi sono sostituite in figura le rispettive luminosità. 55

5.1 Esempio di codice di tipo 2H che utilizza uno schema a 4 colori (2 bits/modulo) per la rappresentazione ottica dei dati. 90
5.2 Esempio di codice di tipo 6H a 4 colori (2 bits/modulo). 91
5.3 Screen relativo alla lettura del simbolo di figura 5.1 su htc tattoo. . . 92
5.4 Screen relativo al funzionamento dell'applicativo su htc tattoo per il simbolo in figura 5.2. 92
5.5 Screen relativo alla lettura di un simbolo di versione 8L a 4 colori su htc magic. Si noti come la dimensione sia inferiore al pollice quadrato. 93
5.6 Screen relativo alla scansione di un simbolo a 16 colori in ambiente desktop, ottenuta tramite scanner Epson Dx6000, che è stata decodificata con successo. 94
5.7 Risultati prestazionali relativi ai tempi medi di decodifica, espressi in millisecondi, per le diverse categorie di codici. 96

Bibliografia

[1] Microsoft Research, *"High Capacity Color Barcodes (HCCB)"*, http://research.microsoft.com/en-us/projects/hccb/, Maggio 2010.

[2] D. Parikh, G. Jancke, *"Localization and Segmentation of a 2d High Capacity Color Barcode"*, pp. 1-6, January 2008.

[3] International Organization for Standardization, *"ISO/IEC 18004:2006, QR Code bar code symbology specification"*, http://www.iso.org, 2006.

[4] A. Grillo, A. Lentini, M. Querini, G. F. Italiano, *"High Capacity Colored Two Dimensional Codes"*, Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT 2010), pp. 709–716, 2010. doi:10.1109/IMCSIT.2010.5679869.

Nota editoriale (2026): Questa voce sostituisce il titolo provvisorio e i dati congressuali preliminari stampati nella tesi del 2010 con la citazione pubblicata definitiva; si veda "Informazioni su questa edizione digitale".

[5] K. Fukuchi, *"Libqrencode, a C library for encoding data in a QR code symbol"*, http://megaui.net/fukuchi/works/qrencode/, 2010.

[6] ZXing Team, *"ZXing, an open-source, multi-format 1D/2D barcode image processing library"*, http://code.google.com/p/zxing/, 2010.

[7] Denso Wave, *"Quick Response Code - printer head density and module size"*, http://www.denso-wave.com/qrcode/qrgene3-e.html, 2000.