Notes on Mobile Programming and Multimedia

Lecture Notes

University of Studies of Padova
Department of Mathematics Tullio-Levi Civita

Penna, Francesco Adami, Nicola

A.Y. 2021/2022, II semester

Contents

1 Cross-platform Frameworks

Nowadays, in the current state of the market we have seen an increase in the environments on which applications may run. In particular, we find four popular platforms, which are the Android, Apple, Windows or Blackberry ecosystems. A company that wants to publish an application must consider the cross-platform feature.
In order to ease cross-platform development, specific frameworks have been created that provide features to ease the creation of multi-platform applications, increasing the support for the developers. This is incredibly time-saving for a developer, since it needs to create effectively just one application, instead of one dedicated to each and every platform. The most known frameworks for this specific development paradigm are React Native, Flutter and Xamarin.
But a question that may arise is wheter or not it is always a good idea to develop cross-platform applications. For example, since Android is executed on so many different devices, even produced by different companies, it is impossible to offer a good emulation in each and every scenario. So we may consider the use of Native Applications, which, again, has cons and pros. Native apps are known for having an overall better user experience, since they allow faster and better performances, while non native applications (cross-platforms) are very limited in using particular percs of a system. For this last reason, Apple always require a native application to be released on their store, since they try to emphasize a better user experience above everything. This obviously causes fragmentation, increasing the costs of development, but also testing becomes way harder with this approach. The table below is good to summarize what we said so far.
Going in-deep, an application development requires four steps:

1. Idea analysis
2. Interface design
3. Application development
4. Store deployment

Choosing a cross-platform development will require executing all four of these phases, while, otherwise, we will need to repeat steps two, three and four for each different platform.
An application that we choose to develop in a cross-platform way will automatically fall in one of the four different classes defined by the **Raj and Tolety classification**, and can be summed as follows.

- Web approach: the final application executes a web service to obtain a native version of the app, like in the instance using a web browser. The instant advantages are that an application as such does not require an installation, but also it is easy to update, since users do not have to manually go to the store. Also, we can use the same interface on all devices (this may not ensure though the same user experience). But, as we said,

we do not need the store, which also can be an advertisement advantage for an app. Also, we always need constant network connection in order to use the application, which will also show to be difficult to test, since it can run on so many different combinations and devices. The changes in user experience may also bring to a low usability, since non-native interfaces, as said, suffer from this inherited problem.

- Hybrid approach: the core of the application is written using web technologies, which are then encapsulated within a native version. Differently from before, this method allows store publishing, as well as a reusable User Interface and allows usage of specific device components, taking the best of both worlds from Web Approach. But we may also see low performances, with a User Interface that does not follow a native Look and Feel of the device (although it is a native applications it does neither look or feel like one).

- Interpreted approach: again, the application is written with web technologies, but, during execution, it has an interpreter that renders the source code for the specific platform it is running on. This allows a native Look and Feel, while also making possible to have a store publishing, and also makes possible to use APIs for specific device components. On the other hand, it makes really hard the possibility of reusing the User Interface, and most of the features depend on the framework chosen. Also, it is possible that the interpreter may have low performances.

- Cross-compiled approach: similar to the previous one, but the interpreter renders the source code during compiling time rather than execution time. This allows to have all components available at all times, as well as a native interface, with good performances and, as usual, store publishing. But, again, the User Interface is not reusable, and the compiling time may be really long for complex applications, making important to have an efficient compiler.

This classification is simplified in *El-Kassan* and other cases, where applications are divided in Native, Web and Hybrid apps.
A crucial element, maybe one of the most important, for the success of an application in the modern days is energy consumption: applications which drain the battery are rapidly uninstalled by users. An action which usually require energy consumption is acquiring data from different sensor, such as the accelerometer, compass, microphone, GPS or camera.
Several authors measured energy consumption of mobile applications and compared the results differentiating between native and cross-platform ones. Thompson and other authors proposed a model driven approach for energy consumption, which require estimating the requirements before the application development. Other methods were used, such as appplications (AppScope, Yoon...) which are used to estimate the energy consumption of each hardware components. Our goal is to compare the energy consumption of different hardware

components during data collection, considering different platforms and different frameworks. The case-study considered were:

- Native Android application
- Hybrid application developed with PhoneGap
- Application developed with Titanium
- Application developed with MoSync using C++
- Application developed with MoSync using Javascript Sensors depend on the API available with each framework.

The results showed that a cross-platform framework is usually linked to a higher energy consumption, even if the framework generates native code. The most expensive task is the interface update, but also data acquisition strongly influences the energy consumption. But this is not a cross-platform only problem, since frameworks showed different consumption depending on the operating system where they were running.
As a summary of what said so far, we can state that cross platform frameworks consume more energy, hence creating lower performances and lower user acceptance. For this and other reasons, hence, native development should be preferred at times, since the framework choice may be too critical, or due to the fact that for complex applications efficient frameworks are still missing. Titanium seems to be the framework with better consumption for the moment, but we should always consider that providing a lousy application is worse than not providing an application at all.

1.1 PhoneGap Cordova

Apache Cordova is an an open-source framework owned by Adobe, and it is the engine below PhoneGap, in the same sense Webkit is the engine of several web browsers. Below we are able to see a first representation of a Cordova Application, and how it interacts with the OS of a mobile device.

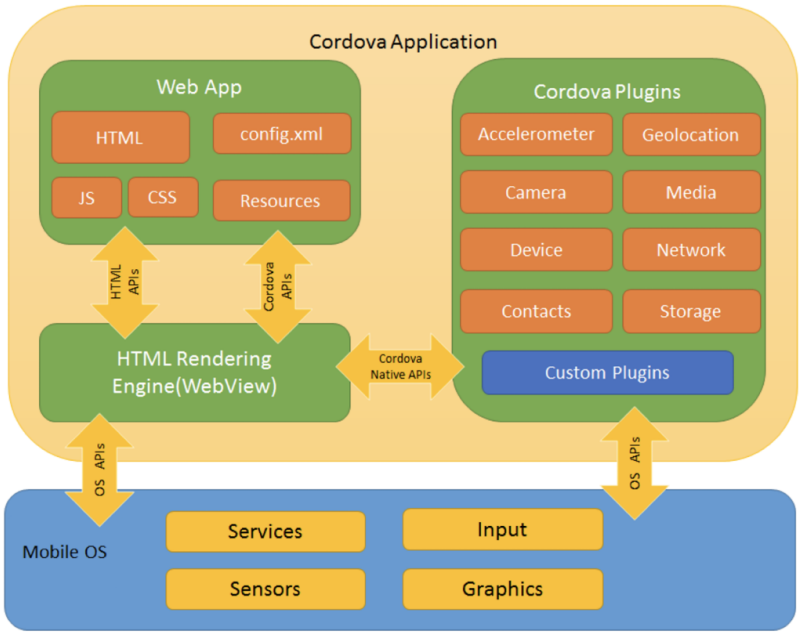


Figure 1: Cordova engine functioning.

As we can see, we find an application defined with web technologies, which, through native or cordova APIs, communicate with the Rendering Engine, for example WebView. This engine is able ot interact with the Cordova Plugins, which make possible to access sensors and network connection. This application is then connected, through the OS APIs, to the actual device, which makes possible to use its services and sensors.
As said, this system, started in 2008, replaces the deprecated PhoneGap ecosystem, with the intention of resolving some open problems such as: developing a mobile application using web technologies, solving the problem of low support of mobile browsers to HTML5 and allow access to different features of the device. The support to HTML5 of the mobile browsers actually solved, partially, some of these problems.
Cordova allows the development of web pages that behave like native applications, named Progressive Web App (PWA), creating a mix between hybrid approach and web approach. These applications are developed using web technologies, but work independently from the browser, using progressive enhancements

(the more feature the web browser provides, the more features the application provides too). They can work offline in a limited way, and can be installed through the store, but they behave as a sort of link. Like every web page, they are responsive to the size of the device they are run on, and secure through the use of HTTPS. Like a web application it is easy to upgrade, but they also support notifications. All things considered, Cordova utilizes the concept of Web Application, coined by Steve Jobs in 2007, and later develops a PWA since it uses new browser functionalities.
Apache Cordova is a hybrid framework which provides tools for testing, such as emulators, and deployment of the final application, with the aim of following the developer from start to end of the creation of the application. The structure of a Cordova application also provides common technologies like a config.xml file, an index.html, CSS and JS custom files, and a WebView container, which works like a browser for the rendering of the application.
While previously it was possible to choose between the PhoneGap desktop application and a CLI version, nowadays the first was deprecated. Altough it contained a *dragdrop* interface easy to use, now it is possible only to use the CLI, which has additional features. An application is created through the `cordova create hello com.example.hello HelloWorld` command, which creates an application with the name HelloWorld, in the hello folder. From this last folder then the command `cordova platform add platform` it is possible to substitute the second platform keyword, and create an application for IoS use or Android. Below we find a compatibility list with the available platforms for each OS.

Platform:	Android	iOS	OS X	Windows 8.1, Phone 8.1, 10	Electron
CLI shorthand:	android	ios	osx	windows	electron
	Cordova CLI Development Platform				
Mac	✓	✓	✓	✗	✓
Windows	✓	✗	✗	✓	✓
Linux	✓	✗	✗	✗	✓

Figure 2: Compatibility between platforms.

1.2 Corona Solar 2D

Solar 2D is a cross-platform framework for mobile applications which uses a cross compiled approach. It has a best use case in the gaming environment, due to its focus on animations, but it can be used to create any type of application. It implements natural interactions in the animations thanks to several physical effects, as well as widgets for a more general-purpose project. Also it is backed up by an extremely active community.
Corona works with the language **Lua**, which is a scripting language used by not only the framework itself, but also by other games which use a native approach, for example the interface of World of Warcraft or Angry Birds. The language itself has three types of variables: strings, numbers and bools. It supports for these data types logical, relational and arithmetical operators. Variables are global by default, while local ones need to be declared with the keyword *local*, and follow the usual scope rules. As for data structures, the most basic are arrays, which are called tables and allow to store data in different position, starting from position 1.
An application is divided in scenes, and Composer is used as the scene manager. These generally represent what is shown on the screen, so for each different section of the application we can create a specific scene. As to give an example, a game may have a scene for the main menu, one for the character selection, one for the settings, and, if it does not require more, one for the game. Composer APIs allow to create, connect, and manage different components (and scenes) of the application. Each scene is a Lua file, where *main.lua* is the one responsible for the start of the application, and then, through composer, we are able to switch between scenes, which are in different files, and create the transition between them. Each scene has four events, for each one an event manager is associated:

- create: adds the object on the screen and the listeners to the events.
- show: starts the timers and the animations.
- hide: stops the objects and the timers.
- destroy: makes it possible to save before exiting.

A scene which will undergo a show or hide event will pass through two different states: will and did. As for the show event, the will happens just before the scene becomes active, and the did just after the scene is presented on the screen. The hide will is just before the scene gets deactivated, and the did after the scene was removed from the screen. These are useful states since they can be used to manipulate information or do operations in between them.

1.3 Xamarin

Let's start with the negatives: this framework require lots of resources from the workstation and has an important learning curve, making it not as easy as other frameworks to start. But it is one of the older frameworks which is still used, making it an important case study. It was acquired by Microsoft in 2016, which made it its default choice for developing multi-platform applications, with the application *Visual Studio*.
Xamarin is interesting due to its approaches, since we find two different ones.

- Interpreted approach for Android and Windows.
- Compiled approach for iOS.

It is a general-purpose framework, and integrates different parts, for example Xamarin Forms is the component utilized for applications which interface with a Server and consist of compiling forms (think of home banking), or Xamarin Native, which is the true multi-purpose part. The core of Xamarin is C, with XML for interface building.
Below we find a configuration of the Xamarin architecture.

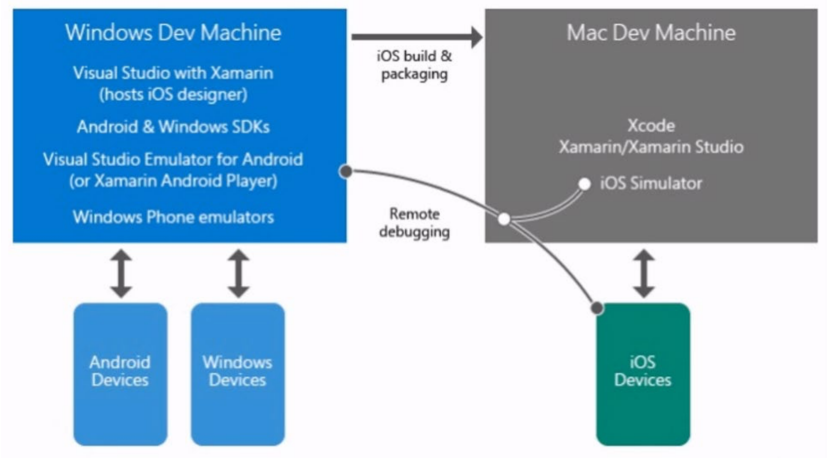


Figure 3: Xamarin engine functioning.

It is necessary to have a Windows Developing Machine, which contact a Mac Development Machine through the network to build the application, which can finally run on iOS. This method, for the compilation of an iOS application, is actually a trans-compilation. Due to its flexibility there is a wide range of companies which use Xamarin for their applications.
The Xamarin Native allows writing code for a single platform, meaning we will need two different native APIs (or even two different projects) for two different platforms if we do not want to write lots of *if* inside one project. This brings lots of efficiency, at the cost of writing more code, even for an interpreted approach.

Otherwise, Xamarin Forms APIs are provided with the aim of working for each platform, providing a more natural lookfeel. For example, if we want to retrieve in Xamarin Native information from the GPS we will call the native API in order to save battery, the interface returned will give us back what we wanted, and the code used in this case will be the same, since, during the compiling phase, the framework will be able to understand which is the correct API to call (either in the interpretation or building phase). This idea does not work for every API, for example in iOS APIs. Xamarin Forms, on the other hand, it's more focused on interfaces which interface with a database in the server, we are creating a sort of webpage, and for this reason it is possible to use a language called XAML.
A question arise in which may be the best approach for developing an application, since we can find choices in the languages to use (C or XAML) and in the end-goal of the application itself (Native or general purpose). We can then say that Form is more appropriate for an application which does not require platform-specific functionalities, and in scenarios where we care more about reuse code. Also knowing XAML is an advantage since it means the programmer will not face an important difficulty curve. On the other hand, Xamarin.iOS and Xamarin.Android (the native versions) are built towards creating a native interaction, so they are good for scenarios which require a native lookfeel. Also, if the application require lots of APIs it is possible of take advantage of the automatic conversion of the compiler, making it easier to develop native interfaces.

1.4 Flutter

Flutter is the youngest framework, making it still a fresh solution for developing cross-platform applications. The first version of it was presented in 2015, while the first 1.0 release dates to the end of 2018. This framework is provided by Google, making it also pretty solid in the sense of versions continuity and APIs creation. One of the latest versions made it possible to develop, alongside with iOS and Android applications, also Windows[1] projects and web applications: basically Flutter is the first edition of a framework that allows to write one code base which can run on virtually anything. The framework utilizes a **cross-compiled** approach (*trans-compiled* according to El-Kassan), meaning we will not obtain an APK which can run on the device, but rather a directory which we will need to open on XCode on iOS for building the application and sending it to the store.
So, we can run the framework everywhere, but in order to create a iOS application we will still need an Apple computer.
The language used is *Dart*, and it is used for each type of application created: the results show that the applications, in particular for the web version, run efficiently, which is an important advantage of the framework. Consider the case of Telegram, while it is important to have a good mobile application, if a message is too long, or if the user is already connected with his computer, having a performing web counterpart gives the user ease to use the service. Due to this efficiency, there are examples of companies which dropped the React Native framework in favor of this new solution.
As said, the framework allows fast development times with native performances (but there are no studies on the energy consumption of this framework), while allowing expressive and flexible UI. Another important feature beside these ones, is the **hot reload**. Depending on the framework and on the workstation used, the time to build an application can easily vary from short to extremely long for really complex examples: this can be an incredible time loss in the optic of testing the application on the developers end. The hot reload serves the purpose of not needing to rebuild each time the application from scratch, since if we make just some modifications (moving a button for example) either the emulator or the smartphone for testing only change the few considered lines of code. This is useful also to test just some parts of a long, complex interaction of an application. Another quality of life improvement towards developers done by Flutter is the use of a large variety of widgets, with material design and Cupertino style. Performances, as said, are really good since we are using a cross-compiled approach and hence creating native applications, even when we access sensors, and the native aspect lookfeel is helped by the widgets, which incorporate different characteristics from different platforms, such as icons and fonts.

[1] Although this is true for version 2.10 (23th of March 2022) the feature has been up for only two months, meaning only time will be able to tell us weather or not this was a success.

To sum-up, we can see all advantages and disadvantages of this framework.

- Advantages:
  - It is a free and open-source framework.
  - It needs to create a single code base which can run efficiently virtually anywhere.
  - It is easy to setup.
  - Brings lot of help to developers through hot reload and widgets. This is emphasized by the presence of lots of plugins for IDEs and a vast documentation.
- Disadvantages:
  - It is difficult to create animations, since it is a general purpose framework and not thought for games.
  - It requires Dart knowledge.

Flutter gives guidelines to the developers in order to gain control of the whole system they are creating, which is a very powerful but also a risk in the wrong hands. Also serves the purpose to create cross-compiled applications as efficient as possible, while trying to obtain fidelity from developers (thanks to all the quality of life improvement they focus on). Another big focus in Flutter is towards *accessibility*: on the web there are lots of guidelines to help all people, no matter what disability or impediment, enjoy content, but in the mobile world there is more or less no interest in these kind of interactions. This is a strange contradiction, since mobile device are not just for reading, but are used to interact in a large variety of ways, like vibrations or lights, and trough all the sensors. Flutter gave an important effort towards accessibility by adding, for example big fonts, a screen reader and color contrast.
As for Dart, it is an object-oriented programming language, developed by Google. While as for the details we refer to the slides of the course, it is important to note that Dart can be compiled in two different ways, which is just-in-time, which allows to compile a component only when it is actually needed at run time (for example in hot reload) or ahead-of-time, which is the feature that makes Flutter a cross-compiled framework.
We now look at the architecture of this framework. It is really complex, but, as a main idea, it is divided in two components: the engine, which runs on C++, and the framework itself, which uses Dart. The engine is the run time environment which runs the application, while the framework itself is the most important component of the SDK, and is divided by levels. The lower levels allow to develop the simplest operations, and hence provide the operations logic which will then be displayed by the upper levels, the last two in particular, which are the Widgets and the Material/Cupertino ones. In the picture below we find this schema.
The lowest level in the framework is the Dart:ui, which allows communication

Figure 4: Flutter engine functioning.

with the Flutter engine, taking for example a button it renders the shape and its main functioning. The foundation level gives important classes for low-level utilities, for example for network connection. Painting and gesture levels manage the colors or the interactions of the interface. Animation is a component for simple tween (an object which moves from an initial to a final position with a speed) and physics-based animations[2]. The rendering level rearrange the layout as needed in order to simplify it. The widget level implements the functionality of the widgets, for example for a button that can be pushed. The last level, material and Cupertino, implements the widget in the according OS style.
In flutter everything we interact with in an interface is a widget. These are organized in the tree of widgets, which is a hierarchy. Also the layout in which we organize the widgets is, itself a widget. On the tree hierarchy, the first widget is the one which indicates which style it is being used, for example material, then we could find the one which indicates which page we are on, HomePage, followed by a scaffold, which follows a Bootstrap-like reasoning of putting elements in the center, to the left or to the right of the scene. A widget can have a state that represent the element that can change, for example with a button which can be in a normal state or in a pressed one: basically, anything that can be updated via an interaction or an external event is a widget with a state. This can have a structural element (i.e. text, size), a style element (font) and an aspect of the layout (padding, margin). A widget is created via the *build()* method, which is called on the root of the tree and than is activated via a cascade to all other nodes.
When building, and hence loading the elements via a render, the framework distinguish between stateful and stateless widgets: every time an event occurs or a change is made the build() is not called on all elements (hot reload), since it is not necessary to rebuild stateless widgets, which cannot change. Only if the state of a stateful widget has changed the method is called, meaning we are able

[2]Once again, this is a feature that the documentation publicizes as easy and functioning, but for the time being it is easy to see how it can be complex to create articulated animations with this framework. If the focus of the project is on this aspect then Flutter, to this day, is not a good choice.

to save time and energy. Knowing this, we will need to consider two methods, which are createState and setState, the first which creates for the first time the state of an object, and the second, to update it and notify the application that the state is changed, in order to call the reload.
Basic widgets include text, row, column, image, raisedButton and AppBar (the top bar of an application). Also Flutter provides an inspector useful to manage all the widgets present in the application. In the Flutter Engine we find the so called platform channels, which are used in three different types: one for binary messages, one for the message channel and one for the method channel. The method channel, calling a method, will make the Flutter application communicate with the iOS or Android ost, which will provide the specific APIs. As a final note, through Firebase it is possible to install packages for Flutter, in order to reduce the work needed for the developer.

1.5 React Native

React Native is a cross-platform and open-source framework which is based on Facebook's ReactJS (*react.js*) library. It uses JavaScript, in particular in its JSX form, so JavaScript and XML, and it can be used for developing Android as well as iOS applications. Along with the already described Flutter[3], it is the most used framework nowadays.
React is a component-based framework, meaning every different part of the application is a component that is independent and reusable. As for a general definition, we can see components as JavaScript functions that accept input data, called properties (i.e. *props*), and return react elements. More specifically, React uses a JavaScript representation of the DOM, called **Virtual DOM**. Each and every time a component is updated, React creates a new VDOM and compares it with the old one, in order to render only what has been modified. A first, simple example, can be seen below.

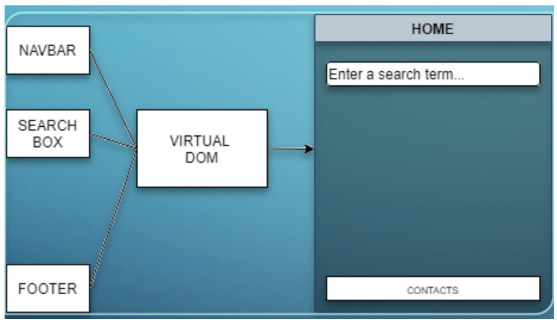


Figure 5: React's Native Virtual DOM.

Properties are used to configure a component when it is rendered, and allow to customize it. They are read-only and influence other components with a top-down approach, meaning if the parent of an element receives a property all its children will also receive it. Along with properties, the *state* is used to keep track of any component data that is expected to change over time, whether it's due to a user interaction, network response, and so on. It can be used to influence components, again, in a bottom-up approach, as the props do.
React Native makes use of the Redux Library, that is used to make the state update synchronous. In order to connect Redux to any application, we will need to create a reducer and an action. The latter one, the action, is an object, with a type and an optional payload, that represents the will to change the state of a component. A reducer, on the other hand, is a function that takes as an input the previous state of the component and an action, and returns as an output a new state. Also, Redux introduces two functions, *mapStateToProps*, and the

[3]Depending on the resources we look at it is said that Flutter received a big investment on the part of Facebook as well as on the part of Google. No matter the truth, the bottom line is that lots of money are on this project, meaning in the short or medium times this will be a solid framework.

mapDispatchToProps.
React's Native components are divided in different classes, where we can find the community components, created by other developers, the core components, installed in the framework itself, and the native components created by the developer itself. Thanks to a bridge, all these component are rendered as native, according to the device's operating system, allowing a native lookfeel both on Android and iOS operating systems.
As a sum-up, we can say that the principal features of this framework are:

- A native lookfeel on every major mobile operating system.
- The possibility to create and use personalized components as if they were part of the React Library.
- An active community available to give very responsive support.
- Hot reloading (check subsection 1.4 to see the definition).
- The possibility to integrate React Native within an already existing mobile application.

1.6 Store Deployment

Once the application is created the work of the mobile developer is not finished, and it does not get too much easier. For starters, there are two main stores for mobile applications: Google Play Store and the Apple store. The policy of the first can be summed as "anything goes", since anyone can publish whatever they want, only if the application contains too many problems or run time errors it gets signaled to Google, which removes the application. There are very few controls in this case, but an entire opposite philosophy is used by Apple Store: here lots of test and debugging are done before putting the application on the store, and it is sufficient that the application does not render perfectly on just one of the many apple devices for it to get rejected.
Whatever the level of testing from the stores, it is hence really important to run deep tests on it. A big part of testing, particularly in the iOS scenario, is also monopolized by acquiring a good number of screenshots for the application, both for smartphones and tablets. In iOS this is a really important step, since it is considered as a big part of the debugging: Apple wants to be sure that once an application is published it can be displayed correctly on all possible resolutions, so on their last variety of products. Once this step is completed the publishing is not over yet, since it has to be considered that the deployment phase could take a long time due to several temporal constraints, not manageable by the developer.
Our in-deep look starts from the **Google Play Store**. It is necessary to have a Google account, and a Google Play developer account, where the latter one is necessary to run the Google Play Console. For 25$ it is possible to have an unlimited duration for the account, as well as an unlimited number of apps. Applications are managed through the Console, where it is possible to see their status, so if they are still published, how they are doing on the store and so on. For applications which require either payments or in-app purchases, it is necessary to create a Google Payments Center account, which is activated from the Console under the "Commercial Account" section. Overall the costs are very low, in order to stimulate developers to publish application, but stop people to publish meaningless projects. The latter point is not really accomplished yet, since the Google Store, as said, does not run too many checks on what's published.
The Google store may change policies, due to the Android system updating, or even due to a change in the legal rules for the applications: for this reason, an app which gets not updated and is not anymore complaint with the current rules gets automatically removed from the Google Store. The application needs to follow hence updates in policies, platforms and operating systems to be able to remain on the store.
Nowadays, Google requires to sign the application before putting it on the store, as it can be seen below.
We see that the key of the developer is used to sign the application created, then successively Google creates an app signing key which is used to sign the application together with the developer key.

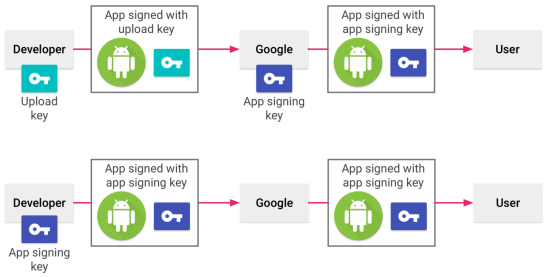


Figure 6: Google Play Store signing procedure.

The process to sign the application is very simple, although it is not possible to use debug signatures. To generate the private key it is possible to use the *keytool* command from a shell, which requires some information, like the keystore password and key password, as well as the developer and company names.

```
keytool -genkey -v -keystore
mykeystore.keystore -alias aliasname -keyalg
RSA -validity 999999 keysize 2048
```

Where `mykeystore` is the name of the app and `aliasname` is the name of the alias. The *RSA* command in this example gives an indefinite validity to the key. To give a better overview, we will now sum up all the required information by Google Play Store to publish an application.

- The APK of the application.
- Name of the app.
- Description.
- At least two screenshots for the smartphones (min 320, max 3840, jpg or png at 24 bits) and at least two for tablets (at least one for 7 inches tablets and one for 10 inches ones), an icon (512x512). It's also needed to require if the application works in landscape mode. The more screenshots, the higher the probability of the application to be accepted in the store. They are also useful to convince the users to download our application among the ones in the store.
- Feature image for the presentation page (1024x500). This is not a screenshot, but is really important for presentation purposes, it can be a factor for the user to choose our applications instead of others.
- Other images or videos not mandatory, such as trailers, screenshots for wearable devices and so on.
- The categorization, so the type and the category of the application.

- Content classification, since it is really important to determine the minimum age for the application. This require a questionnaire, since it is one of the few checks Google runs before putting online the application. In particular we may find questions regarding:
  - Is the application suitable for children?
  - Does the application contains advertisement?
  - Does the application contains in-app purchases?
- Contact email.
- URL with privacy information.
- List of compatible devices.
- Prices and consequently countries where the application is available. Remember that it is not possible to change minds about the pricing of an application: once it has been put on the store as free, it cannot change status and require money before the download.

The Google Store allows to buy different app localization, which are not simple translations but also adaptations to different countries cultures, which can be really useful to promote the application[4]. The store can be used to distribute alpha or beta versions of the application, useful for test purposes. The group study can be a closed group, by email invitation, or an open one, where all the users of the Play Store can participate.
Once the application is published as free it cannot be changed in a version that requires a payment, as said, and the best solution if this change is necessary is to create a new project. The price of an application can be fixed and automatically converted in the currency of other countries, or it can be manually chosen (it's possible for the application to be free in some countries and require payment in others). The freemium version of an application is usually a good solution to let the users evaluate an application before buying it. In any case, Google keeps the 30% of the net amount made by the application.
Timing is a crucial point in the publishing of an application: once uploaded, it is not immediately available, since it needs to undergo tests. In the Play Store, usually, in some hours we have a response. This really fast tests come with the issue that the application may be removed at any time if the quality is not considered sufficient. Also a wrong classification and categorization can lead to a removal of the application.
As far as the **Apple Store** goes, publishing an application requires an Apple Developer Program account, of the cost of 99$ per year. This account is necessary for the signing of the application and has to be done by the user or by the company who publish the product. This costs are much higher with respect to

[4]The example of Nike trying to sell shoes with the characters *Nike Air* which resembled the word *Allah* in arabic country and got a large backlash is still an important case study to this day. Consider really carefully to invest in a localization.

the Play Store counterpart, and it is done to further discourage people to put on the store applications that do not respect the Apple standard. The main line here is the fact that Apple is a company which speaks to wealthy people, since its products are not as well-spread as the Android ones, hence they require a standard. The process of checking whether the application follow these standards are, consequently, much longer than the one for the Play Store. Moreover, developing an application for iOS requires a *Mac* computer, since it needs the Xcode IDE installed, and this is only available on licensed platforms. This brings up even more the costs for iOS development. There are workarounds nowadays, in the sense that it's not always necessary to directly use Xcode, but the program for building the app will use it nevertheless. A big part in publishing an iOS application, as we will see later, regards the screenshots needed.
The certificates that are needed to sign an Apple application are way more complex with respect to the Google counterpart, as it is able to see in the picture below.

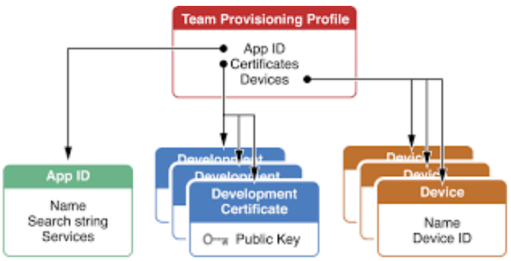


Figure 7: Apple Store signing procedure.

Apple provides different certificates, all of which are needed to sign the application, these are:

- Developer certificate: it is the developer signature of the application.
- Application ID: a certificate linked to the application, it serves the role of defining the name.
- Device: it is a certificate linked to a single device, used during the test phase for deployment of the application on a device that is linked to the developer who signed the application.
- Provisioning Profile: the final application is signed with this profile, which collects information from all the previous certificates. This profile can be on development if the application is on a test phase, or on distribution (or production) if it is ready for store deployment.

A company can be linked at the same time to multiple profiles and certificates at the same time.
The procedure to create the certificates and sign the application require, as

said, an Apple Developer Program account. Then, by going in the profile page of the program we can select "Certificates, Identifiers Profiles", which makes possible to create first an Application ID, and secondly a Distribution Provisioning Profile. Once the latter one is created it is possible to download it and add it to the key-chain (operation possible only on an Apple computer). If the developer certificate was not already installed on the used computer, in order to build the app, it is necessary to download it and add it to the key-chain. Once created and installed all the certificates, it is possible to finally create a `.ipa` file, through Xcode.
Again, we will now provide a list of mandatory information needed to publish an Apple application. Before, we remember that, if we want to publish an app which require a payment, it is necessary to add bank coordinates under the section "contractual, fiscal and banking information". An application can be free, for payment, or with in-app payments. A free application can have in-app payments, but a free application cannot become for payment and viceversa. Like Google, Apple keeps 30% of the amount.

- Platforms on which the application can be used.
- Name of the application.
- Primary language.
- Packet ID, which is the ID chosen while creating the Application ID.
- SKU, which is a unique ID, not visible to Apple, defined by the developer.
- What follows is required to be filled once created a record on the App Store Connect:
  - URL with privacy information.
  - Subtitles, which are optional.
  - Category and classification, which are important as much as is Google Play Store, and it follows the same rules. It is possible to choose two categories and one or two subcategories.
  - License agreement.
  - Screenshots and promotional videos.

At all times it is possible to save a partial draft and continue later, since the process is pretty long.
Screenshots, as already said, is a critical part in Apple approving procedure. This may require a lot of time, since screenshots must be taken for all the platforms the application can be run on, and for each application each and every part must be shown. This is done to check if the application can successfully adjust to the different interfaces and resolutions provided by Apple. Also, having more screenshots is useful to attract more users and convince them to install the application. Also, the screenshots require precise dimensions and can be

only in png or jpeg formats. To help developers, Xcode simulator can be used to create the screenshots for a device that we do not own.
Other information can be added in a second moment, since are not mandatory to publish the application. These are the followings.

- Application description and keywords, which are practically mandatory to fill. This is because they are necessary to place the application in the ranking of the store, and especially necessary to have a good one. Keywords cannot be modified in a second moment, so be really careful about this decision, since they are the words users will search to find our product.
- A URL for support and customer care.
- Icon, 1024x1024 pixels. As for the description and keyword this should be considered mandatory by developers, since it's fundamental for the user to identify the application at first glance. Also it should have the best possible definition.
- Copyright information.
- Version number.
- App review information, regarding, for example, the timing for app release, which can be immediately after the approval or decided by the developer.

Among all these information Apple checks first of all that the application works, without containing malware or other unauthorized content. If a particular category of users has been selected, it checks if it follows the standard required. Remember that tests are made manually, which means it could require up to two weeks if every goes well the first time. If the verification fails for a single small error a following verification may require significantly less time, while a run-time error sets back the times quite a lot. After the publishing, App Store Connect allows to monitor the application, providing information about the number of downloads and crashes, the latter one only if users provide consensus.
A step before all of the ones described in the chapter so far should always be the one of publicizing correctly our product: we should put the user in the position to choose our application instead of another one in order to have success. And to do so we must follow each step described so far correctly, like having good screenshots and icon, categories and keywords, in order to be attractive and easy to find for the users. Once installed, a good application must be used not only the first time it is installed, so another big factor in publicizing the product must regard reminding the user to continue to use our product. This can be done through a careful use of push notifications, which are as useful as annoying. A successful application:

1. Gets installed by the user through good positioning and publishing.
2. Gets opened the first time leaving a good impression.

3. It is able to get back the users for an indefinite number of times.

This last consideration brings us to the last point of the chapter, which regards what happens **after** the publication of the product. The work is not finished at this point, since application age, and in order to remain interesting we must provide new releases, that can be used to fix bugs as well as to provide other functionalities. Android applications suffer another issue related to the monitoring app usage: as said, Google does not run deep tests on the functioning of an application, meaning it is possible to publish an application which does not function properly, and hence need a monitoring of ANRs, or Application Not Responding. Again, crashes are listed only if allowed by the users. In this sense, new versions can also solve these types of errors.
What is really important for an application to have success is to optimize the App Store stand. When an application is first published it has a bigger possibility to have a high ranking, since the stores give points to the newest products. Then, the application begins to age, losing points: for this reason it is important to optimize the score, since a higher rank means more visibility in the stores. This mechanism clearly resembles the one of Search Engines[5], meaning that out of 80-100 results the only ones that get interest and interactions are the first three: it is not just important to appear in the results, but to appear in the first three. A big difference with Search Engines though is related to the fact that the scoring algorithms are not known to the public, to prevent situations like *Google bombing*, meaning we have to do assumptions on how to score higher our product. Some of the most important aspects though were reversed engineered, meaning we have pieces of information while trying to rank higher our application. These strategies are called ASO, App Store Optimization, and they were build by checking if Search Engine Optimization strategies worked on stores too. They are divided in two main groups.

- Onsite strategies: really similar to SEO strategies, it defines methods that regard all modifiable information of the application. In particular, we find:
  - Keywords, both inside the name and ones defined when the app is registered to the store (this is why they are so crucial).
  - Presence and type of icons.
  - Number and quality of screenshots.
  - Presence of the preview video.
- Offsite strategies: similar too to SEO strategies, it work on factors that cannot be modified by the developer himself, such as the number of downloads by the users (*inlinks* in Search Engines). Some of these are:
  - Number of downloads.

[5]Further notes on search engines rankings and hints on mobile stores rankings were also given in the *Web Information Management* course by professor Marchiori.

- Download evolution: the store expects a high number of downloads in the first day an application is published, which will gradually lower over time. If the number of downloads is constant over time the rank is higher.
- Number of installations.
- Number of uninstalls.
- Number of evaluations and its mean.
- Ratings evolution.

Trying to influence in a dishonest way these rankings will usually lead to a ban from the stores. The stores are able to recognize behaviors like the installation of an app in mass from a company, for example.
To get a more precise overlook on how these strategies work, we first take as an example the Google Play Store. Again, we have no information about ranking, but, in this case, the company is the same as the one which implemented the *PageRank* algorithm, meaning we can assume some characteristics will be similar. Keywords for example are excellent if they are contained in the title with less than 20 or 25 characters. Also the brand name should be inserted in the title, only if it's well-known, otherwise, **only** in the Play Store, it can be useful to put a keyword about the content in the name of the application. Keywords are really important in the application description, both in its short and long form. To help this choice, Google provides the Auto Suggest in the Play Store, that shows the most used keywords. As far as the screenshots goes, higher number and quality of screenshots increase the ranking, since the users will be more likely to choice the application. An easily recognizable icon also serves this purpose.
The situation in the App Store is somewhat different, while remaining similar in the main guidelines: even in this case keywords are important, but in this case they are inserted in a field with a maximum of 100 characters. If the application is multilanguage, even the keywords must be multilanguage. These are separated through commas, and it is important to not add the category to the keywords, since it becomes meaningless space. The name of the app should not go over 23 characters and must avoid terms that recall the application content, as explained before. The description can be of 4000 characters maximum, but the suggestion is to use less than that, while including keywords[6], differently from Google Play Store where not adding them is a penalty. In the description remember that the first sentences are the most important. As before the icon, visual material, number of downloads and ratings are critical, but consider that ratings are important *even if* not positive. A huge number of ratings, even with a negative mean, is better than a low number with a really positive one.

[6]Apple declared to use keywords only in the specific fields, but it does not hurt neither is illegal to put them in the description.

1.7 The iOS Platform

Apple operating system changed drastically over the last years: the first complete example we can see is iPhone OS 2, back in 2008, which was re-branded in iOS in 2010. This was the single biggest upgrade up until that point for Apple mobile devices, since, other than the single updates, from that point on there was a switch in the behavior of the company itself: starting from iOS, we saw a major update every year, which adds each time new features. In 2019 iPad OS was also released.
Differently from what we will be able to see in subsection 1.8, device segmentation does not affect as badly iOS as it does for Android devices, but the biggest issues for programmers in developing compatible applications are *layouting rules*.
XCode is the official IDE for developing iOS applications, and is hence mandatory to produce an application for any product in the ecosystem. In order to work with it, it is necessary to have an Apple's hardware which executes MacOS. The IDE receives itself a major update every year. One of the latest news on this topic regards the unique way to build an application without MacOS, which can be done via the *Swift Playground*, an iPad application that gives the possibility to create, and even publish on the App Store, a product. We must note here although that Swift Playground is a simplified tool, so for the best final result XCode is always suggested.
As for the applications themselves, they were originally written in Objective-C, which was written for Apple/NeXT by Brad Cox in the early '80s. It is a dynamic language which takes inspiration from Smalltalk, hence it prefers sending messages rather than calling functions. It doesn't support namespaces, since there is a manual memory management. Nowadays the language was defined as deprecated, and hence is not anymore a safe choice for building an application. To solve this issue, in 2014 Apple announced *Swift*, a language with the principal purpose of becoming the reference language for building applications in the Apple ecosystem.
Apple allows several native frameworks, but the fundamentals are the followings.

- Foundation: it specifies essential data types, collections, dates, time calculation, text processing, networking and so on. It can be seen as the standard library of Objective-C. While the standard library of Swift implements the fundamental data types, other interactions remain in Foundation.

- UIKit: it is the framework which allows to build an event-driven and graphical UI, since it manages the life-cycle events and scenes. It provides utilities to manage interactions and gestures from the user. It is also the Apple way to promote, implicitly, a Model-View-Controller paradigm. It is an imperative UI framework for the Apple ecosystem.

- SwiftUI: probably the most important leap forward since the creation of Swift, which evolved with the aim of enabling SwiftUI. It acts as the main

way to write interfaces across all the Apple ecosystem (tvOS, watchOS, macOS, iOS/iPadOS), with the help of localization, dynamic type and accessibility. This framework also declares the layouting rules that need to be followed:

1. Parent Proposes Size for Child: when a view is firstly created, the root view offers the text (or whichever object) the entire safe area of the screen.
2. Child Chooses its Size: since text-only (or whichever object) may require lots of size to draw its content, a parent cannot force the child's size, it must respect the child's choice.
3. Parent Places Child in Parent's Coordinate Space: if the child goes too much far out of the parent's control, the root view must put the child somewhere.

We will now analyze the most important tools to develop in the best way possible an application. Firstly, we find the App Store Connect and the Developer Portal, which are the portals used to setup and configure all information about the application developed that will be needed for publishing. TestFlight is the main way to distribute the applications to the testers, and it has a dashboard on the App Store Connect. Fastlane allows an automatic creation of screenshots and metadata (which we described how crucial are), as well as code signing and private keys management. As an interesting fact, it is the only third-party tool which became a standard *de facto*. Finally, there is XCode Cloud, annouced in 2021 and hence still in beta testing.
To conclude our overview, we note that there are also two important packet managers in the Apple ecosystem:

- CocoaPods: a third-party package manager to overcome the lack from Apple, which, in the past, was the standard for managing dependencies. It is an automated solution programmed in Ruby.
- Swift Packet Manager: born after a long wait to substitute CocoaPods, it has the main advantage of being integrated with XCode. The packages are defined in a manifest named `Package.swift`.

1.8 The Android Platform

The biggest motivation behind our study of the Android mobile ecosystem is related to its market share: in particular, there are two companies sharing the whole mobile market, Android and iOS, where the first has a 85.5% spread. Not only in smartphones, Android also has a big popularity in wearable, televisions, or even in smart cars. As a further note, Android is the most used OS in the whole world: and this is said factoring *all* operative systems, like Windows, OS X, Linux, and so on.

Android started in 2003 as a startup as a project to enhance the OSs for mobile devices: understanding this opportunity, Google, in 2005, bought the idea, and started to use it with a Linux kernel behind. This was the turning point of Android, up until the first release in 2007, which created a big consortium of companies supporting this OS. In 2008, we had the first mobile device published with Android native. From then there were several releases, until the first really stable one, Android Cupcake in 2009. From version 1.5 Google decided to name each version with a dessert name, up until Android 9 Pie, where they decided to use just the number to identify a new version. From Android Pie 9 to 10 there was a huge switch in the Android ecosystem: they took the opportunity of the change of the name to revise the logo and change the APIs, improving in the best way possible of the system. This switch alone was so big, and we can see it with a simple piece of data: Android 10 was released towards the end of 2019, as of January 2021 the 43.13% of Android devices still used the version 10. But not everything Google touches is made of gold: Android 10 also tried to add foldable devices, with multi-window support. This was not the revolution that someone may have expected, since even nowadays it is difficult to see anyone with this type of device. This was a bad idea from the start: the users had to re-adapt to a completely new interface, with different screen sizes and two windows instead of one, while for the programmer it added even more difficulties in the screens management. On a lighter note, Android 10 saw the introduction of 5G support, and a big switch in privacy, since, at the times, the general public began to be interested in these matters. This was done through a greater control over data permission options, and the introduction of *Randomized MAC address*.

This brief history of Android should hint us a big factor about Android: it is a continuously shifting world, and this can be a problem for developers, since it makes difficult to stay updated.

And to further prove that, around the same times, just a year later in 2020, Android 11 was released. The script remained more or less the same, with Android gaining an even bigger portion of the market, solving privacy issues and introducing new features and APIs. More precisely, they put a big focus on accessibility, which is still today a big missing point for Android (this is the reason why the market of devices for blind people is wildly dominated by Apple). The privacy and security measures were further detailed on the permission stand point: it was possible, from this version onward, to add a 'one-time permission' for an application to gain data from the user.

We finally come to Android 12, which was published as "personal, safe and effortless", all while changing everything. This was actually true, since the interface got lots of updates, which is again, good for a user experience until it gets confusing, somewhat bad for a programmer trying to stay updated. Another focus on accessibility was put in this version, but a big piece missing is still the one of screen readers, which Apple implemented really well a while ago. We will now try to understand, starting from its structure, why it is a problem for the programmer to add so many versions of the OS. Android is actually open source, which is an amazing for a lot of points of view, but at the same time it means that each company can modify the source code to make it more personalized. Also, as we hinted, Android has a Linux-based software stack. These two characteristics bring an enormous issue of Android OS: **device fragmentation**. There are too many versions of Android, personalized by brands, like Samsung, LGE, Huawei, and so on. Another fragmentation issue is the one related to the version. Android phones are more accessible to anyone, even in developing countries: cheaper phones although won't allow newer and more updated versions of the OS, meaning we see a big issue of **version fragmentation** in the market. Nowadays, there are hence way more less GSI, generic system images, which are pure implementations of the Android project.
At this point, the question arise naturally: which Android version should we, as programmer, target. We should certainly consider compatibility with previous versions given what we said, but without forgetting our primary target for the product (which then help to the find the minimum version compatibility). Remember that the more versions we need make compatible, the more the code begins to be complex, with lots of *if* clauses. Fortunately, Android Studio help this decision, since it makes possible to decide the primary target version as well as the minimum target for compatibility. This comes with a very useful schema, the one shown below.

ANDROID PLATFORM VERSION	API LEVEL	CUMULATIVE DISTRIBUTION
4.1 Jelly Bean	16	
4.2 Jelly Bean	17	99.8%
4.3 Jelly Bean	18	99.5%
4.4 KitKat	19	99.4%
5.0 Lollipop	21	98.0%
5.1 Lollipop	22	97.3%
6.0 Marshmallow	23	94.1%
7.0 Nougat	24	89.0%
7.1 Nougat	25	85.6%
8.0 Oreo	26	82.7%
8.1 Oreo	27	78.7%
9.0 Pie	28	69.0%
10. Q	29	50.8%
11. R	30	24.3%

Figure 8: Cumulative versions distribution.

This schema shows us, for example, that if we put as a minimum target version 5.0, we are covering 98% of the market. It is a really helpful tool to design the best application possible for the needs of the project.
We finally created a project, but we still need to see some hints on the functioning of the Android OS. As already said, the base is a Linux Kernel, where everything is based on Java: the actual communication through the system lower levels and with the kernel although must be done either in C or C++. This means that a really precise sensors usage requires knowing the C language. A general use of Android will not require though this level of precision, hence all the communication between layers is done via a Java API framework. The last level is the System Apps, which are not the applications we create or install, but rather the system applications, like camera, SMS, calendars and so on. A further level not as a part of the architecture are the installed applications, accessible via the store.
An Android application, in it's native form, can be written in Java. Nowadays, although, Google is really pushing *Kotlin* as the main language for developing Android application[7], in this sense lots of resources are being pushed towards this goal. Obviously, Kotlin is a way higher level language, so usually an be preferred, but a lower level language, as usual, creates a better grasp of the overall code. The application developed will be compiled with the Android SDK tool, which creates a `.apk` application. This will be the file used by Android devices to install the application.
Each Android application lives in its own security sandbox, protected by various security frames:

- The Android OS is a multi-user Linux system where each application acts as a different user.
- The system sets permissions for all files in the application, so that only the user ID assigned to that application can access them.
- Each process has its own virtual machine, so an application's code runs separately from other applications (all run in different Linux processes).

All of this starts from the principle for which each application, by default, only has access to the components it needs to do its job, and no more. This is done to create a secure environment in which applications cannot access parts of the system for which it is not authorized explicitly by the user.
There are components, on an Android application, which are important to know in order to understand better how to program in the most efficient way possible. Components are fundamental blocs of an Android application and are used as an access point for the system or the user. There are four of them, which are activity, service, broadcast receiver and content provider. We do not know to use all four, but it is important to understand them in order to know what to use.

[7] Google was very straighforward about this. In the span of two years they went from a Java and Kotlin approach to a Kotlin-first philosophy. Both Android Studio and the documentation websites state that Android applications are built with Kotlin.

- Activities: an activity is anything which a user interface, like a button which can be clicked, or even a template. To keep it simple, it is everything related to the graphical part of the application.

- Services: these are functionalities that do not require graphics. These can be in background of foreground. A background service is the GPS functionality in an application like Maps, while a foreground service is the music from Spotify, which is detected by the user, but not visible graphically. There are also bound services, which is the service that provides an API to another process, so, simpler, an application which gives a service to another application.

- Broadcast Receiver: it is a component that allows the system to broadcast a message to lots of applications. A simple example is when the battery is going down quickly, and the OS can send a message to running applications to notify them to save data and stop working. This in the end may produce a notification, but the user still does not interact with this component.

- Content provider: depending on the application, this component may not be so useful. It allow the application to recreate a database done in SQLite. If the content provider allows it, other applications can query or modify the data. A simple example is the contact list, stored in a database like storage, and made available to other applications through the Content Provider.

The components are implemented in such a way that any application can launch another application's component. If the components are in different application the class `Intent` will make possible the communication while respecting the security standards precedent defined. Inside the same sandbox each component can communicate with each other without issues. Since the components act as an entry point to the application, we won't require anymore a "main-like" paradigm: it is possible to contact the entry points and start the applications in different manners.
We hinted the `Intent` class as the main responsible for communication between classes: to activate three out of four of the components an instance of this class is called. To activate the camera, for example, the user can click on the icon component (an activity) which will create an instance of Intent that turns on the camera.
If we use Android Studio to create an Android application we will find files organized by the following groups: *manifests*, which contains the AndroidManifest.xml file, *java*, which has the source code separated by packages names, and *res*, which are all non-code resources. The `AndroidManifest.xml` file is really important and acts as the root of the project, since it describes essential information about the application. Among others, it is important to specify in here the app's package name, the components the application will use, permissions and hardware and software features. Android Studio creates a default version of this file.

Finally, an application for Android can be built in Android Studio, which utilizes *gradle*, an advanced build toolkit. The built APK can then run for testing on an Android device connected to the PC via a USB drive, or on the Android Studio emulator.

2 Mobile Design

A big issue in the development of Mobile Applications is related to the way of designing good, likable applications.
The User Interface design for mobile applications must consider several aspects that differentiate them from the ones designed for PCs. First of all the difference in the size of the devices, as well as the operating system on them. But most importantly, they differ in the interactions: while on the computer we utilize a more precise mouse, on mobile devices we consider touchscreens, sensors, and, in general, more natural interactions.
Users of mobile applications are used to touchscreens, since, nowadays, they are present in many different situations of our lives. More importantly, Apple in 2011 sold more iPads in one year than all the other products in 20 years, but still there are many websites and mobile applications that do not optimize interactions with touchscreens, making this problem really actual in our times. More generally, the market of tablets saw a big increment of sales over the years, which make possible to distinguish interfaces in:

- Smartphones: from 3 to 8 inches, use as interactions touch or gestures
- Tablets: from 9 to 12 inches, use as interactions touch or gestures
- Laptops: from 13 to 18 inches, nowadays can both use touch or gesture and cursor or keyboard
- Desktop: from 19 to 28 inches, use as interactions cursor or keyboard

2.1 Designing a good user interface

When we consider touch interfaces one of the most important things to keep in mind is the reachability of of different components, since it is directly connected to the way we hold a device. Since half of the population tend to hold the smartphone in one hand, and the tablet with two hands, consider the picture 3.1.4. This configuration for interacting with the device present two main problems: the first is the gorilla arm problem, where the user doesn't want to interact with vertical interfaces for long period of times, since its arms tend to get tired. This is an even bigger problem with the increasing size of tablets: 88 percent of tablet usage occurs while seating and, more generally, they are kept on a surface most of the times, meaning they are more similar to laptops rather than to smartphones. This causes another problem: laptops are usually hold on surfaces, but it's way easier to move a mouse in a precise way rather than using fingers that require a higher effort. All of this takes us to a first consideration, that it is preferable, on touch supports, to group controls together to avoid tiredness for the users.
Another important consideration that we can gather from 3.1.4 is the **Thumb rule**. While designing an interface, it is important to identify the most frequently used controls, and put them in the comfort zone, or thumb zone, the

Figure 9: Smartphone comfort zone.

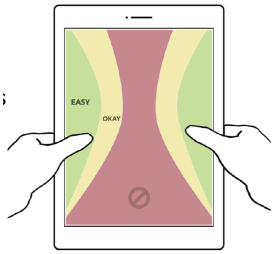


Figure 10: Table comfort zone.

one underlined in green. Also, more delicate controls like data modifications or even deletion should be put outside the thumb zone, to avoid unwanted edits.
In classic website design the priority of the interface is given to interaction elements on the screen, like buttons clickable by a mouse. Touch interfaces change the interaction tool, meaning this paradigma cannot apply anymore: the mouse cursor is tiny, while hands are bigger and less precise. It is hence important to calculate the encumbrance of hands, since important data must remain visible at all times. This generates the *Content Always on Top* rule, which forces to leave the content in the center, moving controls above, below or on the sides of the interface. An exception to this rule can be made for applications like Instagram, where frequent data modifications may occur, in those case the controls must be inside the comfort zone. Since there are many different possible and different OSs, main rules can be summed this way:

- Android: controls must be on the upper side of the screen.
- iOS: controls must be on the lower side of the screen, but consider the new generation of devices, since they have the new mechanism for accessing the home menu.
- More in general: the controls should always remain on the lower side of the screen, but interactions like swipe, or the use of floating trigger buttons should be considered for frequent operations.

On this topic, floating trigger buttons are useful in the case where the design of the OS gets priority with respect to the application design. An example could be the bar to share an article on a news website. This tend to be on the lower side of the screen, since it occupies less space, leaving the content on top. But, since we are on a website, on the bottom we can have the instruments of the search engine. Floating trigger button help prevent this, making easier to hide content and expand it only when it is needed.
What we have said so far is related mostly to smartphone development, since

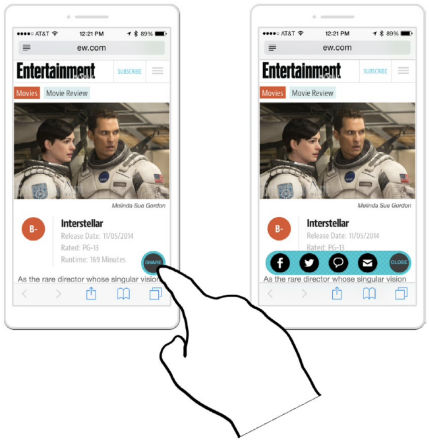


Figure 11: Floating trigger button.

on the tablet case the rules change almost entirely. With bigger screens it is difficult to have a unique overview, and this cause the eyes to move from top to bottom. This teaches us to keep interaction buttons at the top or on the sides of the device, and that, if an element controls the content it must be below or on the side of it, to avoid unwanted modifications. Also, as we said, the bigger the screen gets, the more precision is required, and hence physical activity during the interaction. This means that is crucial to reduce the number of needed interactions for an action, and, when possible, to group together interaction elements.

The change in the medium cause a different translation of known interactions, like the one of *hovering* over an element with a mouse. With exceptions of proprietary solutions that include the use of Styluses, the hover event is not available on touch interfaces, meaning there has to be another solution. The one adopted is that the first tap represents the hover event, while the second is interpreted as the real click.

On the topic of differences between desktop environments and mobile ones, we can consider the consequences of differences between fingers and cursors. According to Google, 83% of websites provide interaction buttons too small to be used with fingers. This gives us an idea also of what are good control sizes: overall, it is a good idea to increase controls dimensions if an error requires more than 2 interactions, 5 seconds, or a context switch. Also, the minimum size to use for a button is 7 millimeters (44px, 1% missed targets), which has to be increased to 9 millimeters (57px, 0.5% missed targets) on big tablets. Also, as seen on 2.2.1, the size of a button can also be influenced by the position. Another important rule is for crowded interfaces: the proximity of the elements is crucial, since the more the elements are close, the more they have to be bigger. Also, to avoid errors, small elements need to be distant from one another. For example, buttons of 7 millimeters must be at least 2 millimeters away. Generally, crowd

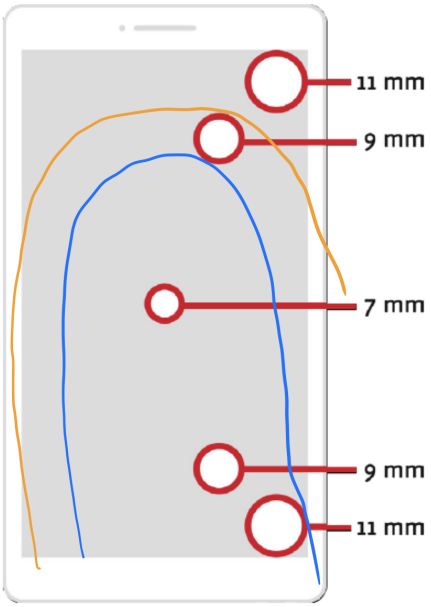


Figure 12: Size of a button in function with its position.

interfaces should be avoided.
In general, a good interface is the one that provides only what is necessary at that moment, which are also called **Just-in-time interfaces**. In this case, the main operations should be available and selectable from a list which shows primary information, and further details should be showed with an interaction, following the *progressive disclosure*. This is a useful approach to make the provided information more clear. A good example of this is the weather app native of iOS.
Just-in-time interfaces bring up the problem of the number of taps needed to interact with an interface: it is important to distinguish quality taps and garbage taps. The first type are the ones that add new information or complete a task, while the second ones could be eliminated, or, at best, substituted with a gesture. Overall, a tap should be added only if it provides a better interface organization.
A big issue in mobile design is related to long pages. For example a website designed for a desktop environment and not adapted to the mobile world could fit three times in the window of a smartphone, requiring horizontal scrolling to be visited. An easy fix to this could be the *carousels*, which should be used with particular attention, or, better, avoided. They cause a loss of overall vision, they make difficult to grasp the connection between different objects, since users do not understand what comes before or after. Instead of forcing the user to make several swipes for finding the information, it is better to ask for a single tap to open a page with more details. A good example of this better approach is the

application Zalando Privè. A question could then be when the carousels are effectively useful. Some examples could be:

- Linear data: like the weather app, in which the user knows what to expect.
- Casual browsing: in the case of pictures or with a slideshow, they work even better if they are inserted in a context in which users know what to expect (shouldiuseacarousel.com).
- To break up very long forms: in this specific case, the advance cannot be automatic.

Another topic on the problems of interfaces adapted to the mobile world is the one of long forms. A study proved that 1 in 5 cases in which an online shopping is not completed, it is due to the excessive complexity of forms necessary to complete the steps. In these cases, each field required makes the difference: decreasing from 4 to 3 fields can increase interactions by 50%. On mobile interfaces we lack the tab control, meaning each field requires one tap more than in the desktop environment, hence it is important to not ask for more than the needed information. The absence of a tab button is related to the absence overall of the keyboard support, which we need to consider since the user is generally not fast at typing. So, to help them, we should always:

- Provide the correct keyboard for a specific input.
- Prefer a list of buttons to a menu if this one is too short.
- Avoid too long drop down menus.
- In case of number insertions, give the options of +/- buttons if the number is not far away from an average median.

Lastly, confirmation dialogues were introduced to let users think about their answers, but nowadays they do not work anymore as intended, and just slow down the user. To help this, it is possible to implement a swipe movement for example to answer a call or to unlock the device. This movement is sufficiently difficult to be only intentional while remaining easy enough to be fast. Also, makes possible to undo what did.

2.2 Gestures

The last point discussed in 2.1 introduced the idea of a gesture, which makes a task for a user easier to achieve, without being too complicated. Before describing how gestures are useful, it is important to distinguish them:

- Tap: click for the touch world. Overall, it can be interpreted also as the hover event, as discussed.
- Swipe: frequently used for scrolling, view change or to show hidden panels.

- Long press: used for context menu or to unveil detailed information. It is usually used only by expert users, since in Android and iOS it opens the contextual action bar. It is the equivalent of the right-click in MS Windows.
- Long press and drag: equivalent of the *dragdrop*.
- Pinch or spread: zoom in or out.
- Double tap: also zoom in or out, but is can be used also for other purposes, like selecting an element to which an action has to be applied.

All of these actions should be enough to convince us to delete buttons (unless they are needed). This is also given by the fact that, in real life, we are used to buttons like a light switch, which toggles an action in a different place with respect to where we pressed it. With touch interfaces, in fact, everything should be reachable, so we need to ask firstly ourselves if there is a way to manipulate the content directly before using a button. Buttons also have the tendency to cover important content, meaning removing them could also lead to more usable space in the interface. We already discussed how buttons have an inherited problem of precision in them: gestures improve an interface accessibility since they tolerate less precision, helping in situations in which, for example, the user cannot pay close attention to the interface or when it is needed to have a fast interaction with no errors. Also, big gestures tend to become reflexes, since they build muscle memory and are not based on visual memory. A successful example of why gestures are so important is the game Angry Birds.
The world of gestures is deeply linked to the one of **metaphors**. For example, a card metaphor is something frequently used and deeply understood by users: think of applications to manage business cards, plane tickets, or coupons. Cards suggest to us several interactions, like flipping them, or putting them in stacks. But, in the mobile world, we flip a card by clicking on a button, which is not really manual: this teaches us that is always important to help the user understand and get natural with an interface.
We hence ask ourselves in which way we can help the user: a first good rule is to get help from the real world, in the sense that a gesture similar or equal to the real one has no need to be teached to the user. An example of this good behavior is the one of drawing apps. Even if it is not possible to follow a real world gesture, everything must follow conventions, meaning it is extremely important to not betray users expectations. 2.2 shows an example of application for a list of groceries. In this case, to add a new item between two existing ones it is necessary to drag away the upper and lower one, creating this way a new card. It has to be teached to the user since it is not directly intuitive, but once learned it becomes very natural.
Gestures should overall be easy to understand and to use by the user, but it is also possible to introduce complex gestures that can be used like keyboard shortcuts. These are also called **shortcuts** and are implemented when for an action the user need more than one touch. For these gestures it is necessary to

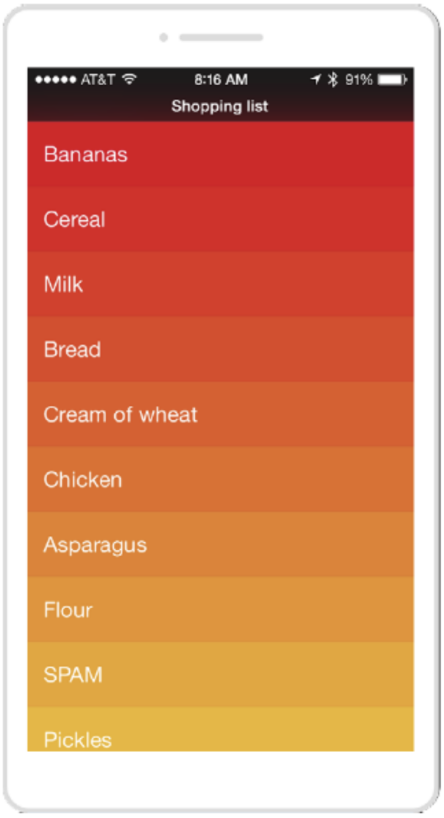


Figure 13: Good example of natural interaction.

think of longer gestures, and it is also possible to use multiple fingers. Overall, this last idea is not always good, since it is sometimes not supported by devices and it is not easy to learn.
An issue related to gestures is their relationship with operating systems, since they have priority: gestures by the OS cannot be used by the application. In Android the OS gestures always start from the sides, but iOS is more complex, since they can happen from anywhere inside the space dedicated to the app.
To solve some problems related to gestures an idea may be the one of introducing

2.2.1 How to teach gestures to the users

If we decide to stick with gestures, we need to ask ourselves how to teach them to the users, which bring problems to the table.
This is a serious issue, since, unlike buttons and widgets, gestures are almost always invisible to the user. Obviously, considering the times we live in, the introduction of manuals is not feasible, since, the first time a user uses an

application, it comes with a precise objective to satisfy. This is not compatible with reading instructions. Some gestures can be trivial to learn even for non-skilled users, but some will have to be discovered by the users. The only solution remains **just-in-time** education. Practically, we do not send the information from the start, but we wait until it is needed.
Together with just-in-time education, it is also important to consider the idea of **skeumorphic design**, which entails representing elements of our application with real-world elements. The most famous example is the cycle bin as the trash icon in a personal computer. Al tough skeumorphic design isn't always suited for gestures teaching, the main concept that is important to remember is the one of the metaphor: we need to find the right metaphor to represent our application and not betray it, in order to not confuse the users. An example of

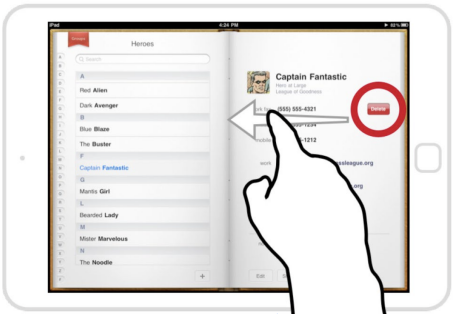


Figure 14: Example of skeumorphic design.

why skeumorphic design is not a good idea to teach gestures to the users is given by the first version of Apple's contact book. In order to make it more familiar to the users, they tried to make it look like a real world contact book. Firstly, the delete button might be in the way of the gesture, and it might be better to move such an important button to a more appropriate space, as discussed. Moreover, the user expects in this case that, if they turn the page, the contact page will change. This is not the case, hence breaking the metaphor.
The biggest issue here is the fact that making interfaces way too realistic can be dangerous, since the users expects a simple concept: looks like =¿ acts like, and if this line of reasoning is broken, the user will be confused. The inverse of this reasoning can stand and it is why, nowadays, we see less and less skeumorphic designs. Excessive realism can have the effect of limiting the possibilities of an interface.
Another idea to teach users gestures is using plain and simple **instructions**. It is a false myth that, if the user needs instructions than the designer has failed to design the interface, we just need to understand when to give the instructions. Overall, a good idea is that it is better to learn something while doing it, since it also helps develop muscle memory. Video games are a medium that solve the problem of the moment in which to teach something to the user really cleverly: in these scenarios the user does not know what to do but learns while playing

mainly through three ways.

- Coaching: the main idea is that telling someone what to do is not as effective as showing them how to do it. In order to do this, tutorials may be implemented, in which easy instructions are provided when a problem shows up for the first time. The issue of this methodology is to understand when a skill is considered learned, since from that point onward instructions are no longer needed. To do this, it is possible to ask to repeat a gesture from 3 to a maximum of 5 times, depending on the difficulty. The mistakes of the users help us understand when a gesture is not learned. An example is when a user stops during the interaction, since that could be a warning to toggle an animation, to show a complete gesture.
- Leveling up: current guidelines for modern teaching suggest not to teach everything from the beginning, but to rather provide a small-steps knowledge. This is the main idea behind leveling up, which teaches only basic interactions at the beginning, and let the users use complex gestures if they autonomously find them. Users will feel motivated to learn more complex interactions when they need them, and in those case they will be thought. For this reason, app must be organized in different levels of complexity.
- Power-up: more typical in video games, they translate in mobile designing of UIs when they provide facilitation to the user. If a user does a gesture frequently it is useful to provide it a shortcut, since it will cause great satisfaction. In this sense, power-ups are gestures that make an interaction faster or easier. An example may be: once a user has tapped 10 times on a button to access a functionality, we can tell that there is a faster gesture to access it, providing an overall sense of satisfaction.

2.3 Other interactions with interfaces

Considering the evolution that smartphones and mobile technologies have achieved over the years, we can nowadays interact with many sensor in our devices, not just the touch screens. Between the various examples of sensors, we can consider:

- GPS, useful, to provide the user's location for applications that include a support for mapping.
- Accellerometer, compass, and gyroscope can be used to recognize movements of the user.
- Sensor like the luminosity one can be used for accessibility purposes.
- The camera can be used to collect data inside applications for translation, QR code reader, or, again, accessibility.
- Fingerprints reader.

2.4 Emotional design

Emotional design is a new frontier of mobile UI design, since emotions are, at times, the best way to teach the user things about our application. As seen in

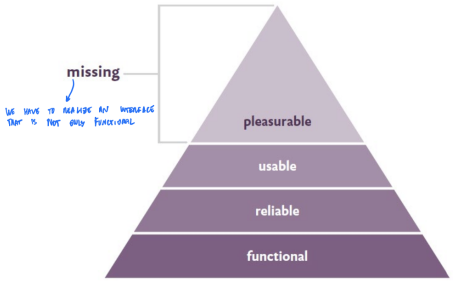


Figure 15: Pyramid of user's needs.

3.3.4, nowadays it is not sufficient for an application to be just functional or reliable, or even useful for it to be successful. A huge part of the success of an application depends on how pleasurable it is to use, since otherwise it will be easily forgot in the stores. To describe in more details the various parts of the pyramid, we can describe applications as:

- Functional: if the user is able to complete the assigned task.
- Reliable: if the system work, in this case every kind of failure is unwelcome.
- Usable: if it is easy for the user to learn how to use the system and its functionalities.
- Pleasurable: if the user experience in pleasant.

In the same way we don't place our hands on a stove in fear of burning them, emotions are essential for memory management, since they can be used as reminder. Again this time we will use elements from the natural world to help us building better applications. Studies have shown that people are attracted to elements positioned in the *golden ratio*. Choosing the best size for the elements in an application may cause a reaction which is not necessarily conscious on the user side, but is judged in a good way. This was used in the design of the user interface or the Twitter dashboard. Another natural element used by Apple was the patent of a status led that followed the human breath, an interaction that is not easy to notice but help internally relax the user.
The Wolfwsvagen *Maggiolino* helps us introduce another topic in emotional design: its enormous success show that users are attracted to objects that are humanized. **Humanization**, something really similar to a human, is typically liked by users. Another good example for this phenomenon are the MM's, designing candies like humans improved the sails of the product, also, in this example, using really big faces is a factor in the strategy of marketing: newborn babies tend to have bigger heads than adults in proportion to the rest of

the body, which is an unconscious information we own as human beings. This, linked with the facts that humans protect the human children, and say yes to them, tend create a positive effect in users.
Another way we can help users sympathize with our products is through the use of personalities, more precisely, a mascot, which should speak, answer and act accordingly to how it is designed. In an application this can be used by making the mascot draw from a random pool of audio messages in a database and execute them while the user is interacting with the application, creating an engagement. Some customers may understand this trick though, making in reality pretty hard to apply mascots to applications. Firstly, before creating such a figure, we must create a user archetype: we have to collect information about the brand image that we want to show, which also include the personality, and the language used by it. Also, how can the mascot get the user attention, which are the emotions users are attracted to.
This last topic is pretty significant, since it deeply changes how a mascot interacts with the users, influencing the success of the application: emotions are powerful, hence need to be used accordingly. There are no strict rules, but powerful tools include surprise, pleasure, preview, exclusivity and rewards. The best example of what preview is like is a movie trailer, which help creating interest in the user. A good example of exploiting users emotion is done by Nike, which allows the selling of a particular set of sneakers only to subscribed members, or starts the selling at a certain time, creating exclusivity and preview. This is, though, a risky strategy, since a person will decide considering pros and cons, but, when these are not possible to measure, will just go out on instinct, and the major obstacles for this sensation are laziness and skepticism. A correct design will help mitigates these problems, but also the use of games (think of Duolingo) and incentives can help the user prefer our option.
For all of these strategies to work we need to check multiple things about the emotions we are using.

- Is the persona created for the brand correct? While using a bank application we do not want have mascots surprised at how poor we are.
- Is our product too similar to other competitors? This was the problem of *Google+*, which was way too similar to Facebook.
- Are user need satisfied? For example, in social networks, if we are able to convince multiple user to stay in my application (or move to mine), since the aim of social networks is to connect with friend and celebrities.
- Is the language correct? It is not always possible to use informal language in most applications, but generally when we have a disruption we should tell the truth directly (the 404 error).
- Is my application still usable, enjoyable and reliable? After trying to invest into emotions to make the application more usable, are we still respecting all the other, fundamental, servises?

3 Multimedia Data and Encoding

The world in which we are used to live nowadays is a hypermedia, which is a conjunction between hypertext (the structure we are used to associate to the Web) and the use of multimedia data. This means that a high dimensional scenario has to deal with difficulties related to the synchronization of different continuous media (video, audio, animations, and so on). But what actually is multimedia?
A multimedia system is a system for storage, integration and management of heterogeneous complex information, to represent the real world through multiple modalities. This implies we need to make space for both text and graphic, or for recorded audio and animations, so completely different ways to treat information. Again, we find ourselves asking what, the, are "media". This term has changed with the evolution of technology, so, while in the visual communication domain it represents the different mediums of communication (papers, radio, television, cinema...), in non-computer based multimedia presentations we can intend as a media an artistic performance. But, in the origins of this term, "medium" used to specify the domain used to represent the information, so text, image, animation or sound, and this is the best way to intend the term nowadays.
Multimedia innovations are not only related to technological aspects as it may seems: for example, the increasing interest in human computer interaction brings problems related to the complex interaction between users and new systems, like, for example, highlighting a link in a video or audio file in a way easy to understand for the user. But also, the inappropriateness of permanent copies: since human beings prefer permanent objects, that we can keep in our hands, how can we use properly audio and video? So, technology is not the only frontier in multimedia studies, we also have to develop competences in methodology, dramatization and psychology (imagine a network with congestion, we need to make the user wait as little as possible). But all these competences leave open problems: a higher expressivity means higher complexity, which implies that the learning curve can make the product hard to use. But also, too much flexibility and variability of exploration can be energy-consuming and confusing.
The bottom line in this topic is that we will apply multimedia strictly to a Web scenario, meaning we need to clarify the differences between WWW and the Internet. The first is a distributed application, that uses the Internet as a technological infrastructure. Internet brings to the table services, protocols, remote storage systems, and so on, while WWW provides the user interface, the applications and the environment for distributed applications. Multimedia systems are related to both worlds: youtube, for example, provides an interface for videos but also is a storage system, meaning these boundaries are not always so well defined. Good news though, multimedia technologies are independent from the WWW, since representation, compression and transmission of information do not always need to be shown.

3.0.1 Media Classification

We can define a media classification on mainly two characteristics:

- Content: text, image, audio, video.
- Media dynamic: diffusion and media usage, which can be
  - Static media: images and text, so information that do not change over time.
  - Dynamic media: audio and video, so information that change over time.
  - Temporized media: live streaming, so video that can only be seen in a particular moment, meaning information changes over time in function of temporal constraints.

Image, audio and video, no matter what, all require an encoding and compression, which will influence the final quality of the object. We will study encoding algorithms, which have different advantages in term of size of the compressed object, the time required to compress it, and the quality obtained. There will always be a trade-off, meaning usually if we are able to have a low encoding weight we will have lower quality. Not considering advantages and disadvantages, we have goals on what a multimedia encoding algorithm should do: it should reduce the size of persistent data, optimize the transmission time for continuous data (videos on youtube!), decode fastly encoded information in a symmetric or asymmetric way, and control the quality of the decoded data.

3.0.2 Media Properties

Considering a dynamic media classification, we start from **static media**.
In this scenario, users read information with his/her own timing, meaning data transmission does not impose temporal deadlines, if they are in a reasonable limit. The information considered is persistent, meaning it can be seen multiple times. Examples of this are text and images. Data size, in this case, is not a problem. A higher size means a higher compression time, but this is not a problem for the user, which has to interact with it. In most cases, information diffusion and usage of static media take place in separate moments: the user can download and display information separately, or save them on the local storage of their devices, or, again, cache them in their browsers.
In **dynamic media** the user must follow information evolution in real-time, meaning the diffusion can be delayed, but information meaning may change, for examples in online videogames. Obviously, information is not persistent, and a good example of this are animated presentations, youtube and video files. Differently from before, here size is a problem, since the information is frequentely big, and should be used in a predefined time, like during a video. Like before, we can download and consume later the information, but more commonly we are in a streaming scenario, where proprietary solutions are usually preferred. Finally,

we have **temporized media**, where the user must access the information following the timing schedule of it. The diffusion must follow a media isochronous timing, meaning audio and video need to be synchronized and they must be reproduced at a defined fixed frequency in order to be understood. Information is not persistent, and, in most cases, it cannot be used twice, like in real-time streaming videos. The size of the information in this case is proportional to the reproduction time.

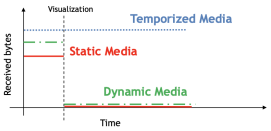


Figure 16: Received bytes for the media types

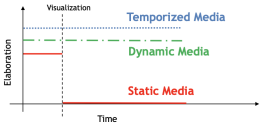


Figure 17: Elaboration needed

3.0.3 Information Compression

Storage compression can be achieved in two ways, by either reducing the number of bits necessary to encode the information, or *entropic compression*, or by reducing the information that need to be memorized or transmitted, or *differential compression*. Compression can also preserve or not the original data, so lossless, or lossy.
Ignoring these types of compression, the steps are three:

1. Transformation: information data are transformed in a domain that can require less bits to represent data values.
2. Quantization: obtained values can be grouped in a smaller number of classes and with more uniform distribution. This is not a reversible step.
3. Encoding: information is encoded based on the new classes and values, together with the encoding table.

Note that quantitation is the phase that actually reduces the data during the compression, meaning it is not present in the lossless compression, while lossy follow all the three steps.
For the reminder of the section we will focus on lossless compression techniques.

3.0.4 RLE, Run Lenght Encoding

When data contains a lot of consecutive repetitive values, there is a lot of redundancy that can be canceled (a good example of this is a block of the same color in an image). Long sequences can be substituted with a repetitive symbol, its value, and the number of repetitions. This compression is efficient for sequences of at least three equal values. In the RLE, the first packet has a bit set to 1, in

order to show that that is the RLE packet, followed by the number of times the character is repeated. An example of this can be:

BGFDDDDDDDDDIJUPPPPPHYTGBUYYYY INNNNNNGHHHHHHHKPPPPPPP →BGF@9DIJU@5PHYTGBU@4YI@5NG@7HK@7P

RLE is suitable for artificial images, or ones that have few details. On the other hand, images with shades or high contrast are not suited for this protocol, since it could just increase the number of bytes needed. For text too this is not the best solution, since there is no language that repeats 3 letters.

3.0.5 Shannon-Fano algorithm and Huffmann Code

Before describing the two techniques, we need to define **entropy**: it is a measure of the quantity of information in the bits flow that has to be compressed, meaning it is the number of theoretical bits that need to necessarily be encoded. Entropy of an alphabet S can be expressed as:

$$H(S) = \eta = \sum_i p_i log_2(\frac{1}{p_i})$$

Where p is the probability of finding the i-th element, while the logarithm represent the minimum number of bits necessary to encode the i-th value, so that it can be recognized among other elements. Based on the probability of finding a specific value inside a sequence, we can create an entropy encoding, which encodes a value with a different number of bits based on its frequency (more frequent symbols will use less bits to be identified). Shannon's Theorem, on the topic of lossless compression, states that the lower bound of the average code length (i.e. the compression rate) is the entropy of the information source, meaning

$$H(S) \leq L$$

. This means we cannot go below the number of bits of the original file, since otherwise the compression would not be lossless anymore.
Starting from here, we can describe Shannon-Fano algorithm, which creates a binary tree with a top-down method. Each node is a symbol or a group of symbols. Nodes are sorted based on the number of occurrences, since we assume that we will need a different number of bits to represent 'E' and 'Z' in an alphabet. Below, an example of how the word 'hello' gets compressed. The

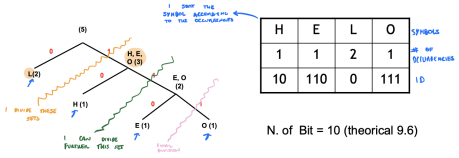


Figure 18: Example of Shannon-Fano algorithm.

first step is dividing L, which appears two times, from H, E and O. After that,

we can arbitrarily divide these letters, since they each appear one time. In the tree, each right branch gets the label 1, while the left 0. The labels are assigned based on the path that need to be followed in order to find a letter in the tree. Huffmann Code is a very similar method of compressing information, but it creates a tree with bottom-up method. At each step, we take the two nodes with lowest probability, creates a subtree with probability equals to the sum of probabilities of each nodes, and goes up to the root. The final result is really similar to the Shannon-Fano one. In the figure we can find an example. Again,

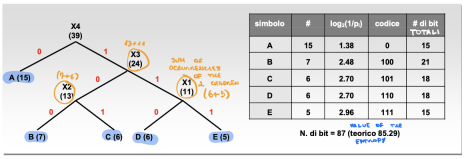


Figure 19: Example of Huffman code algorithm.

the code of each symbol is the label of the path from the root of the tree. The final number of bits that are needed to encode a given string can be represents as a sum of two quantities: the number of total bits needed for a single character, which can be seen as the number of bits used for the code times the number of occurrences of the symbol (so, for A it is 15*1, B is 7*3 and so on), plus the bits needed to represent this table. In case of single characters we will need 5 bits (needed to store a *char*) times the number of symbols, in the example 5*5, plus the bits needed for the codes, so, in the example, 3+3+3+3+1, hence 15. Finally, the example will require 87 + 40, so 127 bits.
Since the codes are built from the leaves we will not have codes that are prefixes to other codes, implying it is possible to decompress from any point in the file. The drawback of these methods are the representation of the table, if we try to use it for all the possible colors in an image we will have 60000000 rows, which is not manageable.

3.0.6 LZW

LZW is a method that dynamically builds the vocabulary of the symbols using fixed codes for the sequences with variable length, meaning we can have a code representing two or three symbols at once. Hence, we avoid having to memorize the table. Below, we can see an example of the encoding phase. The main line is that if the string we are currently looking at in w, with the new one k, already exists we put wk as the new w, in order to form a new symbol for the dictionary. This process begins to be efficient after usually 100KB, while the beginning is a bit slow. An important note on the encoding is that we output just the symbols, and not the codes, since we do not bring the whole dictionary with us in the decoding phase. The corresponding example of the latter one is shown below. The decompression recreates the vocabulary while expanding the text, which is given back as an output. We read a character and see wheter we

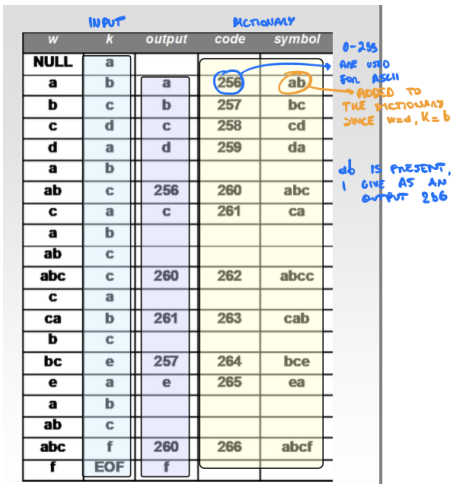


Figure 20: Example LZW encoding.

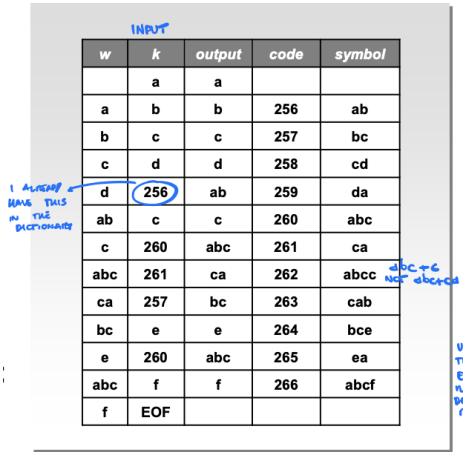


Figure 21: Example of LZW decoding.

have an entry for it in the dictionary: if we do, we output it, we add w with the first character of this entry to the dictionary, and we set the new w as the entry. The size of the table is dependent on the number of bytes we are using, meaning if the dictionary is full we remove an entry and restart, usually the first one. Usually, LZW is more efficient than Huffmann code, that is optimal just for entropy coding. In this case, LZW compresses a variable number of characters and not single symbols.

3.1 Images Encoding

Once again, before starting the in-deep analysis of image compression, we begin with an introduction on the fundamentals about images, and the way we, as human beings, perceive them. Because, the best way to understand how to make a better compression algorithm is to first understand what we can and cannot perceive of an image.
An image is an area with a defined color definition, whereas a digital image is a bidimensional matrix, where each point has a chromatic information. The digitalization of an image comes in two steps: firstly, we have the spatial sampling, where we define the resolution and the number of pixels, then we find the chromatic quantization, in which we focus on the colors. If we use 24 bits we can represent 60000000 colors, which is a true color representation, but we may not need all of them.
The human eye has, first of all, a sensibility, which is the visible spectrum, and is the electromagnetic spectrum visible to the human eye, and can go from micro meters to one kilometer. A wavelength is between 380 and 750 nanometers, which are not visible to the human eye. The retina of the eye is formed by rods and cones, where thge first are responible for night vision, while the latter are of three types, and respond respectively to red, blue and green colors. Our eye is more sensible to the green rather than to the red and blue. On the topic of colors, there are two different models to rapresent them, and they are both based on the combination of primary colors:

- Additive color, RGB: the one used for monitors, where the presence of a combination describe the color.
- Subtractive color, CYMK: the one used for printers and similar devices, where the absence of a component define the color.

In both models, combining the three primary colors we can obtained other colors, like cyan, magenta, yellow, or black.
But another way to conceive colors is through psycophysics characteristics, and this brings to a further level of distinction:

- HSV: hue, saturation and intensity values, the hue (i.e. tonalità) is described starting from the red, going through the yellow, then blue, and back to the red, in a circle like shape. Both saturation and intensity, which is the luminosity, are described on a grey schale, and are hence a percentual value. It is based on the perception of the human beings of shades and gradients of color.
- YUV, YIQ, YCbCr, which separate information about luminosity from color information.

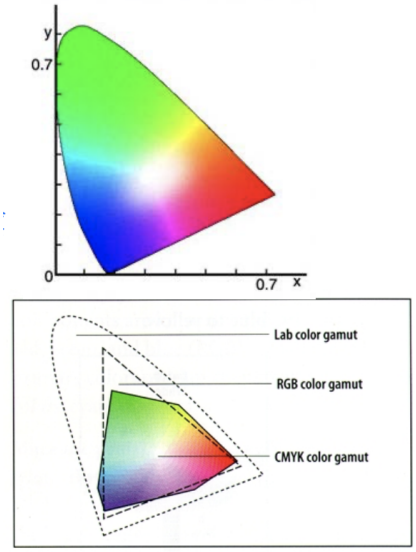
The bottom line of these models is that when we take out luminosity we gradually understand less about the image in question, meaning we cannot only base ourselves on the colors to understand what we can lose and perceive in an image.

Until now we described models only basing ourselves on the devices used for the visualization, monitors, printers, TV and so on. But a theoretical model, non used for practice, can help us see the full gradient of colors that we can perceive, and this is the **CIE XYZ**, which is a tristimulus representation system, and describe not a color but directly what our cones perceive. We describe colors based on three coordinates, in a system like so:

$$\begin{cases} x = \frac{X}{X+Y+Z} \\ y = \frac{Y}{X+Y+Z} \\ z = \frac{Z}{X+Y+Z} \end{cases} . \tag{1}$$

Which implies that the sum of the three components is 1. This shape is called the horseshoe curve, and rapresent pure colors, like it can be seen in picture. The

Figure 22: Horseshoe and confront with other models



line of purples connects the extreme points of the shapes and represents colors which does not correspond to a wavelenght, and hence cannot be perceived. At the center of the diagram there are some reference colors.
After the analysis of possible different models to represent colors, we go back to the properties that a digital image has, which can be divided, as said before, in color depth and resolution. As for the first, we see monochromatic and gray-scale images, which require respectively between 1 and 8 bits. Then, there are images with indexed colors, which are usually 8 bits, so 256 colors chosen between a palette or a Color Look-up table. Finally, we find true colors, which use 24 bits and represent 16 million colors, going further, with 48 bits, we see high definition. Hence, with 8 bits we cannot represent colors, just grey-scale, since a colormap, which is the bottom line, has 256 colors. The resolution

of an image, instead, is limited to the more common sizes of screens for Web applications, since images can be encoded in different resolutions, and even with miniatures useful for previews.
We now show different types of image encoding, starting from the MS Windows native format for images.

3.1.1 BitMap, BMP

This format is used to represent images with either a *palette* (example of colormap) at 1,4 or 8 bits per pixel, or with natural colors, using 24 bits per pixel. This format is lossless, since it uses the RLE algorithm to compress images. It is hence particularly suited for artificial images, or images with several sequences of equal pixels, meaning it is not a good solution for photographic images.
A palette is obtained by dividing the RGB cube by a standard number for each channel: so, with 256 colors and 6 steps, we get 216 equally spaced colors, where the other are freely chosen. The pro in using this strategy is that, in the communications between the same OSs we won't need to share the palette, since it is native, but different OSs will also have different palettes. Stupid as it may seem to the reader, a good example of palette is the original color system from Windows '95 *Paint*.
As we said, BMP does not compress the original image, and is a lossless compression format, but this is not really feasible with the real-world requirements, since the memory occupation depends on the color depth of an image, its size and the color table needed for the decompression: this quantity can range from 18.75 Kbytes to 2.25 Mbytes. Compression helps out in strongly reducing these requirements, brining it down, in JPEG, to roughly 300 Kbytes. We are focusing on this aspect since the transfer time for not compressed images is usually not acceptable even with fast networks connections, going up to 30 seconds needed. But the intervals of time needed are even below 4 seconds when we talk about an image that just need to be visualized on the screen (the good news is the quality requirements are also lower in these scenarios).
This brought the study of multimedia to create image formats, independent from the platform and the producer, and which do not require a license.

3.1.2 Graphics Interchange Format, GIF

First off, GIF is the only format we will cover which do require a license. But it is also the first standard for image transmission over networks, which were original faxes. It index 256 colors, which are very few, and use the LZW algorithm for a lossless compression. It is also used for interlacing, transparency and animations. Good use cases for GIF are simple images with few different colors, or artificial images. The lossless compression has also the advantage of a fast decoding. Another pro is the fact that animated images are usually engaging and make messages more visible for advertisement purposes. Also, it supports transparency, which allows a easy integration with the background.

3.1.3 Portable Network Graphics, PNG

It is an extensible format, with lossless compression, portable and though for colorful images. It has no patents or licenses. A color can be represent with a variable number of bits, from 1 to 16, meaning it is a universal solution for indexed colors, gray-scale images and true color images. Transparency is achieved through the use of an optional alpha channel. In order to make easier transmissions through the network it uses an interlacing method, which is shown below.

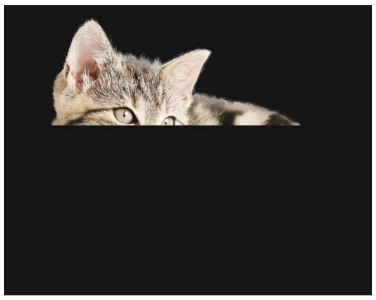
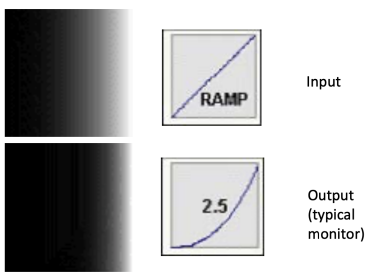
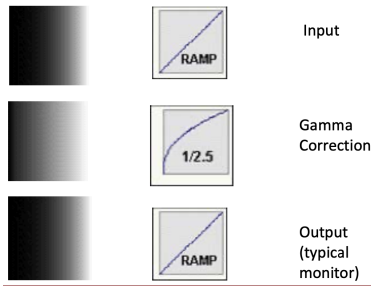


Figure 23: Half data without interlacing

Figure 24: Half data with interlacing

To the right we can see that we have not the best resolution possible, but we can define an image instead of having half of it shown.
PNG also uses a mechanism of **Gamma correction**, since, in the real world, input images tend to have less shades of black with respect to the usual monitors. To help this, a gamma correction function is applied to an image, in order to help contrast a bit the black shades of the monitor, as it can be seen below.





PNG has the same uses as GIF, while not requiring a license. It is able to achieve equal or better quality than the JPEG format, but with a lower compression. The presence of an alpha channels of 256 transparency levels helps differentiate the transparency effects. It is an extensible format.

3.1.4 Joint Photographic Experts Group, JPEG

This format is the standard for photographic images compression since 1992, and it tries to compress images by understanding which data are redundant, hence cannot be seen, in order to remove them. It is hence an example of lossy compression, overcoming the limitation of the entropic compression. Again, the bottom line is that the smallest details of an image can be suppressed without losing useful information.
The JPEG encoding is done in six phases:

1. Image preparation: we switch from RGB to YUV color model (or similar).
2. DCT : for each 8x8 block in the image, a DCT (discrete cosine transform) is applied. The DCT moves an image from the domain of color to the one of frequencies. This generates a new 8x8 matrix, where the element (0,0) is the dominant color, while the other values rapidly tend to 0.
3. Quantization: each coefficient is divided by a weight defined in the quantization table.
4. Substitution: the value (0,0) is substituted with the difference with the same value of the adjacent matrix, in order to obtain low values.
5. Matrix linearization: the obtained matrix is covered diagonally, to get the adjacent values equal to zeros.
6. Compression: RLE and Huffman are used to encode the resulting list of values, creating a subsampling quantization.

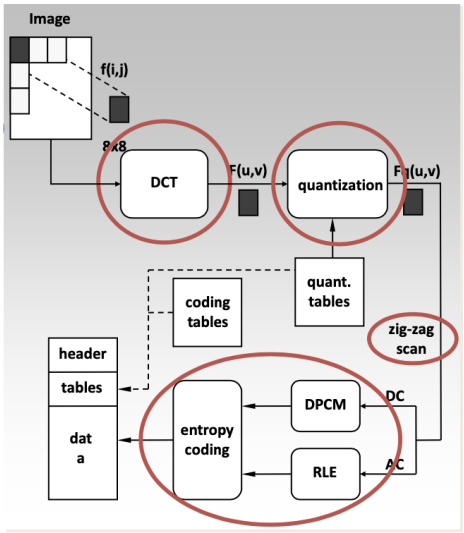
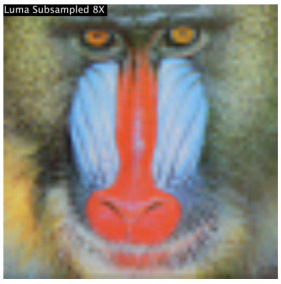
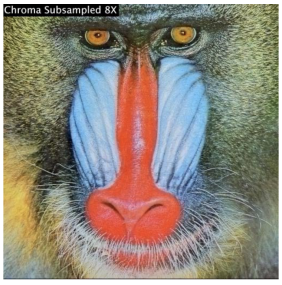


Figure 25: Sum of the JPG compression.

This process lose information in the preparation (about the colors), during quantization (the image becomes blocky), and some during the DCT (since computers use something similar, but not actually real numbers).
If we lose information during the preparation process, why do we **switch color representation** then? The human being perception is more precise for luminosity, rather than for colors: the representation of luminance information must be more accurate than the chrominance one, and the two components have different resolutions. So, from the YUV color coding we consider the Y (luminance), then U and V (chrominance) get encoded as differences to the reference colors. An important detail is that we consider only U or V component, obtaining half the data with respect to the original component. To get a better understanding, consider now the three images below, where the first is the original one, the second the one where we downgrade by 8 times the luminance, and the third one where we do the same with the chrominance.





We now discuss the **Discrete Cosine Transformation**, or DCT. After we lose the data from the previous phase, we consider each 8x8 block in the image, and apply the DCT to break up the input signal into its different components: our aim this time is to represent the dominant color. Each coefficient in the DCT defines the weight of the relative frequency inside the image, meaning the biggest one will be our dominant color.

$$F(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{N}\sum_{y=0}^{N} cos[\frac{\pi(2x+1)u}{2N}]cos[\frac{\pi(2y+1)u}{2N}]$$

This function is theoretically reversible, but since we are losing data due to the fact we are not able to represent R, this is not precise in a real scenario. As specified before, the element (0,0) contains the predominant color, while the other elements a variation of the color. Each of the obtained 8x8 matrices follow this rule, meaning the closer to (0,0) we are able to see fewer details, that increase while we stray away. After this phase we consider only half of this matrices, the bottom half part: this is due to the fact that more details get confused by the users and are not necessary to the compression of the image.
After the DCT we find the **Quantization**. As said, the DCT is (theoretically) reversible, hence the original image can be rebuilt, but the DCT coefficient of the quantization causes the degradation of the restored image. After the

quantization, indeed, we see in the upper part of the image really low values, meaning those are the components we are able to perceive, and we need to store, while straining further from the upper half we find really high value components, that we cannot see. As an example, consider an image of size 10x10 formed by 8x8 components. Using just the first component to represent the image we wouldn't be able to see almost anything. Using the first four of the upper part of the matrix, we could see more, and the use of 8 components would make the situation even better. Using 14 components we wouldn't be able to distinguish the original image and the compressed one, meaning we didn't use 76 components, an astonishing result.
Going even further in the JPG compression we see the **Linearization and Compression**. The main idea here is that, after the quantization, we have lots of zeros in the matrix, that can be summed up with RLE encoding. This is due to the same idea as before: highest frequencies represent details of the image that can be suppressed without losing noticeable information. The 8x8 matrix with the DCT components is linearized with a zig-zag scan.
The JPEG compression has four different coding methods:

- Sequential: each image is encoded with one single scan from top to bottom, left to right.
- Lossless: it is very similar to PNG and hence not considered often.
- Progressive: allows to show the image with low quality at the beginning and progressively with increasing quality. This is implemented through spectral selection or consecutive approximations.
- Hierarchical: the image is under-sampled and JPEG coded, and after the image resulting from the difference between original and rebuild is coded.

JPEG is the best format for photographic images full of colors and shades, which is also its standard use case, and is also useful for precise representation where minor details are not essential. As explained loading and visualization can be progressive (as we will see further). It is also possible to regulate the quality of the image.

3.1.5 JPEG 2000

The question we may find ourselves asking at this point is why we do actually need a new format of JPEG. We were originally able to move from GIF to PNG, then, from PNG to JPEG, where the image, in the resolution 1200x600 for example, could save up to 2000 KB of space. But the whole philosophy of JPEG, as we explained, is to work in the 8x8 blocks of an image. Meaning we can represent lots of details, but if for some reason the process does not go smoothly we are left with a very bad error management: we would be able to see the pixels in an image. Moreover, as we hinted, JPEG itself is not suited for images that cannot afford to lose any information, like for example images in the medical field.

These reasons brought the creation of JPEG 2000, which is the last compression standard for image distribution over Web and smartphones. Its main feature is the fact that is more resilient to lost data, but it is not thought to substitute directly JPEG, rather to have a gradual transition. It is strongly oriented to wireless scenarios, since it focuses on transmission: the same 34,5 KB image, that would require 14 seconds to download with JPEG requires 1,2 seconds in JPEG 2000.
With the exception of the web browser *Safari*, JPEG 2000 is pretty difficult to implement, since it requires a specific CPU configuration, and a plugin in the used applciation. It uses a *Discrete Wavelet Transform* (DWT) that provides an encoding that supports multiresolution without data redundancy, both with lossless and lossy encoding. The bitrate, as said, is way lower than in JPEG, and it supports more compression modalities and color spaces. It allows up to 256 information channels, achieving satellite images, differently from the 3 channels of RGB. To help transmission in disturbed environments (like the wireless ones) it implements a Region of Interest (ROI), which is a place decided a-priori that will use the most bits in the image. It is an open standard, and supports images even bigger than 4 GB, working well with both natural and artificial images. The compression systems used are the wavelet, and the DCT, allowing a full compatibility with the original JPEG. The **DWT** elaborates each color independently, producing four regions, known as tiles, such that each tile dimension is half of the original image.

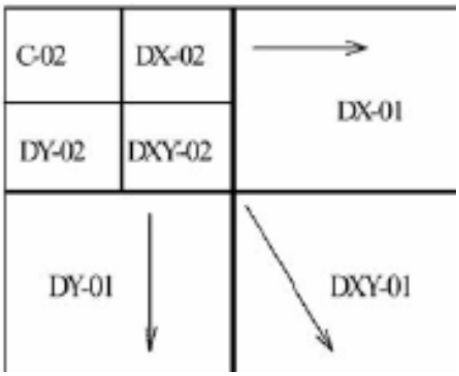


Figure 26: DWT tiles.

As seen in the figure, the tile in the top left is the one with the lowest frequencies, the three that round it contain the details about the horizontal, vertical and diagonal sides of the image, like shown by the arrows. The idea is that the lowest frequency are closer to the first quadrant, while going further we find the high ones.
The **Quantization** in JPEG 2000 works on the base of the sensibility of the human eye over contrast variation. The quantization coefficients are defined based on the sensibility to the associated sub-band. Again as before, we need

the quantization table inside the JPEG 2000, in order to decode it.
The final result of these processes is a better decoding phase, where even few details allow to show images less blocky with respect to the original JPEG. This is also made possible by the **ROI**, a technique that isolates an important image area and encodes it with higher quality than the rest of an image (usually, the background). This method is called MAXSHIFT and is based on shifting coefficients related to the ROI to the highest bitplanes. Below it is possible to see and example of ROI.

Figure 27: ROI example.

Now again, consider the example below, in which we examine the difference of errors tolerance in the two formats, to the left JPEG, to the righe JPEG 2000, for the same image, with a BER of 10^{-5}.

Figure 28: Example of errors tolerance.

Looking to the left we are able to see a white stripe without color in it. That's because of the encoding of JPEG. In the transmission we lost the DC component related to the block that starts at the white strip, meaning we are not able to represent the color in the row. In a similar way, below, we lost information about another DC, but on a lower degree. JPEG 2000 manages to patch these problems, and we can see to the right the image, a bit shady but safe from clear errors.

3.1.6 Hints on Vector Graphics

Up until now we talked about pictorial images, matrices defined by assigning a color at each point, or pixel. Vectorial graphics are described using geometrical formulas that define shapes, color fills and positioning using only mathematical features. A great advantage in this case is that they are really easy to compress being only formulas, we can even use a lossless compression like LZW. Pictorial

images are the result of elaboration from the calculator. It is possible trough this to obtain high precision, since the quality of the image does not depend on the pixel: is is also possible to obtain details enlargement without losing quality (scalability), meaning when we zoom on an artificial image like so we do not see it blockier, but rather are able to look closer at the details.
Vector graphics are good for paper advertisement, because they are easily editable and scalable. Also 3D graphics are a good application, or the construction of fonts and animations for videogames.

3.2 Audio Encoding

As for the chapter about image encoding, we will have an introduction about fundamentals of sound, to have a better understanding of what the human ear can and cannot perceive: this will give us an idea on how to get a better format for compression.
Sound is a longitudinal pressure wave that propagates through a transmission medium. The range of human hearing is between 16 HZ and 22 kHZ. A sound is formed by an amplitude (volume, the intensity of the sound), a frequency (higher or lower sound) and a waveform (the timbre, allows to recognize different types of sound production). Fourier analysis shows that a signal can be broken down into a series of related sinusoids, each one with its amplitude and phase, and frequencies that are harmonics of the fundamental frequency of the signal.
But the first step in this study is the digitalization process of the sound, of these sinusoids Fourier cites. The first method is the sampling, which is a time division, and then we can find a quantization, which is a discrete representation of the signal level. The digitalization of a sound can be shown below in an example.

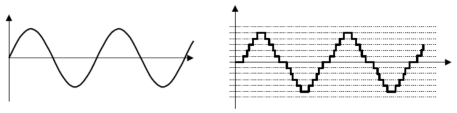


Figure 29: How a computer sees a signal, to the right.

The Nyqyust theorem suggests that if a periodic signal contains no frequencies higher than N hertz, it can be completely reconstructed if 2N samples per second are used. This is a huge result, since it shows we can reconstruct a real signal in a digitalized form without losing any information.
Digital encoding and decoding of audio signals is more problematic than the image encoding: the temporal structure of the audio cannot be modified (frequency), and the audio information varies over time. Also, the required reproduction quality goes beyond the simple understandability of the message. As for the images though, the uncompressed audio strategy is not suitable even for conventional networks, since it would require for FM Radio quality 1.2 Mbit/sec. This is due to its high dimension, since, for example in a streaming scenario, it is not possible to transfer the whole audio file before playback.
There are many different audio formats that have been developed over the years, some of which are:

- Waveform Audio File (WAV): developed by Microsoft and IBM, it is the standard used for audio encoding on PCs, and it does not use compression.
- Audio Interchange File Format (AIFF): developed by Apple, it is the standard for MacOS, and, again, although it supports a compressed version, it is not compressed.

- μ-LAW: standard audio format for Unix, it is also the telephonic standard of the USA.
- A-LAW: european version of μ-LAW.
- MPEG-1: it encodes audio tracks in MPEG-1 videos, meaning it is a compressed format for variable quality encoding. The algorithm works on several steps based on the psychoacoustic principles, and allows three different encoding levels with three different bit-rates. It is used as the cross-platform standard, since several applications for consumers market use it widely.

Coming to think why MPEG-1 is the preferred solution by the market for audio encoding, a first consideration has to be done about the lossy compression it implements. Lossless compression for audio is not feasible, since it provides really low performances: audio data are extremely variable, and it is rare to find repeating patterns in it. Moreover, audio information is redundant, meaning we can control the compression quality to please the human ear. Some concepts that are implemented are:

1. Silence compression: we identify silence as a consecutive set of samples under a defined threshold, and compress it with the philosophy of RLE.
2. Adaptive Differential Pulse Code Modulation: encoding the difference between consecutive samples allow to have a quantization of the difference, meaning we can lose information which is not meaningful.
3. Linear Predictive Coding: we adapt the signal to a human speech model, and transmit the parameters of the model and the differences of the real signal to the model.

But before actually applying these techniques of compression, we need to understand what we can and cannot lose, and we can do it with psycoachoustics elements.
The human ear sensibility varies along the audio spectrum: the maximum sensibility is around 2-3 kHZ, and decreases at the spectrum extremities, but we have to keep in mind the sensibility is deeply influenced by personal factors. Humans perceive sound and voice at an hearing interval of 20Hz to 22kHz, whereas the recognizable dynamic interval, which is the interval between the weakest and strongest sound, is roughly 96 dB. An instrument that helps clarifying an aspect of human earing are the Fletcher-Munson curves, which are displayed below. We can see here the thresholds for hearing and pain, but the main focus is in the part between two red lines: in there, we perceive the sound at the same volume. These frequencies, where there is a uniform perception of the amplitude of the sound are grouped in **critical bands**. Each band has an amplitude from 100 Hz to 4 kHz, and the spectrum can be divided in 25 of them. Doing this, we can consider the human hearing as a series of overlapping band-pass filters.
The main issue now is what happens when sounds overlap: a pure sound can

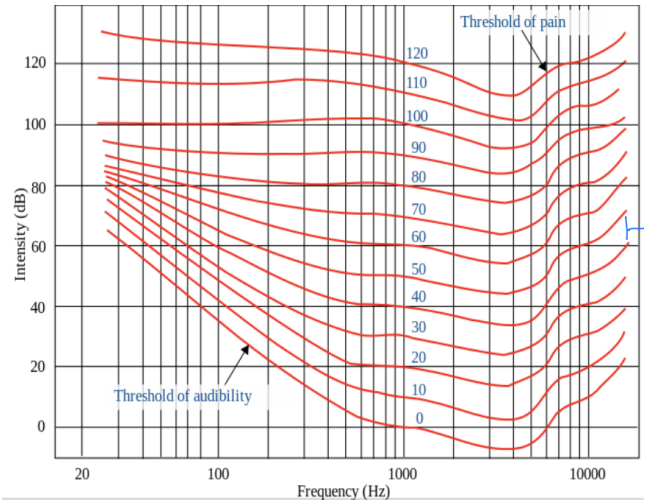


Figure 30: Fletcher-Munson Curves.

mask another with near frequency and lower volume, but, during the playback of a sound at 1 kHz, other simultaneous sounds in the masking interval cannot be perceived. This is different from non-tonal masking, where we involve a sound that has a higher intensity than the masking sound (but the masking one is still able to silence the other).

3.2.1 MPEG-1

MEPG-1 layer 3 (MP3) is the current standard for high-quality audio (music, in general) with compression. The most common bit-rate for MPEG standard is from 48 KB/sec to 384 KB/sec, and, even though its compression level is 6:1, it is able to obtain a result almost identical to the original signal. From 96 to 128 KB/sec, it represents the best quality for consumer applications, since it also supports monophonic, dual, stereo and joint stereo signals. MP3 is a transparent encoding, with a transparency threshold of 2.1 bits/second, from which we are not able to distinguish the original file from the compressed one. MPEG audio compression algorithm works on four steps, based on the psychoacoustic model:

1. It divides the audio signal into 32 frequency sub-bands.
2. For each sub-band, it calculates the masking level.
3. If the amplitude of the signal in the sub-band is lower than the masking threshold, the signal is not encoded.
4. Otherwise, it calculates the number of bits necessary to represent the signal (from 0 to 15) such that the quantization noise is lower than the masking threshold (1 bit is close to 6 dB of noise).

5. Creates the bitstream following a standard format of transmission.

This process, which is only the encoding part, is summarized below: Here we

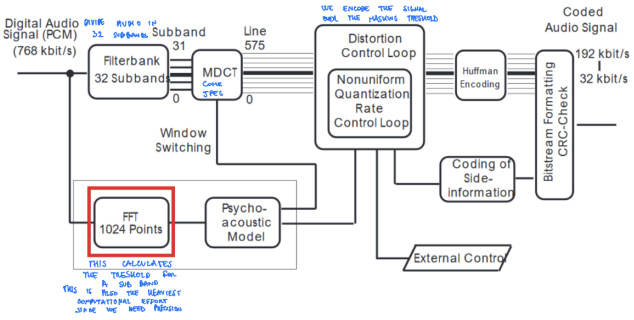


Figure 31: MPEG audio encoder.

can see the components that divide the original audio in sub bands, the MDCT, like the one used in JPG, that is used to check wether or not the sub-band is lower than the masking noise. The latter one is calculated below, in the FFT, the Fast Fourier Transformation, which is the heaviest computational effort of the algorithm, since we need precision. This means that the encoding of a MP3 file is quite heavy on a computational stand-point, while the decoding does not have this section. This is due to the fact that in the decoding effort, we do not need to calculate again the thresholds, making MPEG an **asymmetric compression**.

Consider the following example: the masking level for band 7 is 12 dB, 15 dB for band 9. If the level at band 7 is 10 dB it gets ignored in the compression, since it's lower than 12, while if at band 9 we find 35 dB it ends up compressed. Remember that only the difference between the signal and the masking treshold is encoded in the end.

Another useful application of masking is the one of **watermarking**. This is the inclusion of digital information (like source, destination, copyright, information, and so on) hidden inside multimedia data like images, videos, audio text and even animations. The watermarks cannot be modified and do not have the necessity to modify the enclosing data. Watermark insertion is done in an audio by doing the opposite to the MPEG algorithm: you insert the watermark near high-level signals, such that it gets masked, and it is not distinguishable from the original one.

MPEG-1 at layer 3 uses a bit-rate of 64 Kb/s, and divides the frequencies spectrum in different sub-bands with non equal amplitudes, which are more comparable to the critical sub-bands in the lower frequencies. This is the current most used standard, and following it also MPEG2, in 1994, and MPEG4, in 1999, were developed. The first is the standard for DVDs, and was created aiming at transparent sound reproduction for theaters. The latter consider audio as a composition of different objects, where the user can decide to listen

a composition in different environments, or even emphasize some components more than others.

3.2.2 Musical Instruments Digital Interface, MIDI

MIDI protocol dates to 1983 and provides a standard and efficient way to describe musical events, in the same ways a score is used for real-life instruments. It enables computers, synthesizers, keyboards and other musical devices to communicate each other via a scripting language that codes "events". Sound generation is local to the synthesizers and messages describe the type of instruments, notes to play, volume, speed, effects and so on. MIDI on itself is a pretty complicated system, as the image suggestsm but most of modern soundcards come with all the necessary hardware.

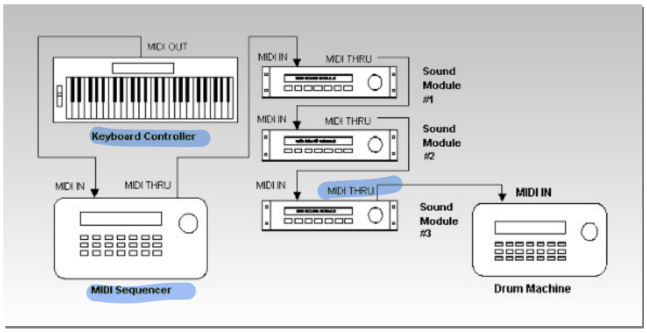


Figure 32: MIDI system example.

We can see above the MIDI sequencer, which is a recording and execution system for storing and editing a sequence of musical events, in the form of MIDI data. It receives data from the input device, allows editing (like speeding up the tempo of a song, not possible in MP3 without altering the frequency itself) and creates the music sending data to devices like sound cards. MIDI has no influence on the quality of the sound during this process, which is entirely reliable on the synthesis device.
MIDI is organized through:

- Channels: allow to send and receive music data, they are a method to differentiate timbres and send independent information, this implies different channels are used for different instruments. MIDI channels start from 1 to 16.
- Tracks: a structured autonomous flow of MIDI messages, like in a piano song, where there are a melody and an arrangement.
- Patch: specifies the timbre produced by the generator, MIDI can contain up to 128 different patches, which contain pitches.

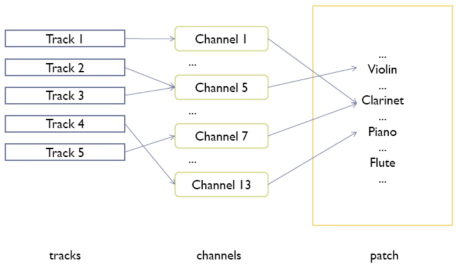


Figure 33: MIDI music representation.

This schema is summirized below.
MIDI messages are divided in channel ones, which describe which not to play (voice) and how to play it (mode), and systems, which define set-up and synchronization information. All of these messages are made by 10 bits, with 8 of them of useful data, one at the beginning and one at the end. The 8 bits are divided in nibbles of 4 bits, where the first represent the message type (channel or system), while the second the number of the channel to which the message is directed: we use 4 bits and MIDI use 16 channels.
Actually, **system messages** do not use a channel because they are meant for commands that are not channel-specific. These type of messages are divided in:

- System common messages: carry out general functions related to the whole system, like setting up a common clock. They are also used for track selection and positioning inside of a song (consider when we fast forward or go backwards in a song).
- System real-time messages: related to real-time synchronization of the different modules of a system, they are used to start or stop the playback of a song.
- System exclusive messages: these are used to extend to the product the MIDI reproducer is in.

MIDI is an efficient standard to encode musical sounds inside web documents, since its compact and does not depend on the waveform of the sound. It is particularly well-suited for background music. But it can only represent traditional western music, and sounds like voice or other acoustic phenomena are not encodable. Also the quality, as said, depends explicitely on the quality of the audio card of the device, which needs a specific architecture to use MIDI in the first place. Also channel and message coding is not completely standard, since it may vary, for example, between synthesizers.

3.3 Video Encoding

An analog video is encoded as a continuous signal that varies over time, meaning it can be digitalized but not further elaborated, due to the bi-dimensionality of the image. A digital video is a sequence of digital images, where we see a continuous color information flow: in this context we see two important phenomena about video. The first, is related to the persistence of vision: this traditionally refers to the optical illusion that occurs when visual perception of an object does not cease for some time after the rays of light proceeding from it have ceased to enter the eye. The second is a frequency which defines the frame-rate and is described as the frequency of which an intermittent light is received as to perceive the signal. The combination of these phenomena allow us to perceive a video.
A first example of digital video can be seen in an interlaced video, where each frame is divided in two types of rows, odd and even, which are shown alternatively: according to the number of rows we can achieve from 25 to 50 frames per second. Others definitions of digital video can be drawn, depending on the types of signal they require:

- Video with separated components: each primary signal, like RGB and YUV, is transmitted separately, allowing a better color reproduction due to the absence of interference between signals. This, altough, requires high bandwidth due to the precise synchronization.
- Composite video: luminance and chrominance signals get mixed in a single carrier wave, and, differently from before, we see interference.
- S-Video: chrominance signals are mixed in a single carrier wavem while luminance ones are sent separately.

The transmitted videos trough these signals have properties, which are related to different characteristics. First, the color depth, which is usually a true-image, due to the recording. Then, the resolution, which depends on the standard of the video, while the chrominance gets undersampled, remember that higher resolutions require more space, like the example of 4K videos which need up to 5 GB per second. Finally, the frame frequency, which depends on the region in which videos are transmitted: PAL uses 25 frames per second, while NTSC roughly 30 frames per second. The minimum to avoid the perception of snap movements is 15 frames per second. Usually NTSC is referred to north America and Japan, while PAL to Europe and most of Asia. In this context we find the **sampling**, which is a ratio that describes how much information we have about a video: with a 4:4:4 we have all the information at any time for each pixel, while as for the 4:2:2 we find, every four pixels, 2 pixels of information about chrominance, 4 about luminance, and 2 about Cr and Cb values. Other combinations are 4:1:1 and 4:2:0.
Being a digital resource, a video will have a memory usage, which, if not compressed, require a considerable amout of storage: high definition television (HDTV) requires a bit-rate that can be higher than 1 Gb per second, meaning

we must compress data: 1 hour of MPEG-1 video with a VHS requires roughly 600 MB, which paves the way for a lossy compression paradigm. Firstly, in a similar way to what did in the previous chapters, we must encode intra-frame and inter-frame information, where the first refers to the compression of single images, the second the compression of data due to redundancy. Also, in this scenario, we must focus on eliminate spatial and temporal redundancy. The memory usage highly influences the transfer time of a video over a network, which has the same problems as the images transmission, combined with the fact that it is a continous medium, it must follow a constant frame-rate and the loading time must be compatible with the reproduction time. On this latter point, a download and play solution, nowadays, is not always applicable, which is why we find ourselves in a streaming scenario. This means that, rather than a server giving all the data to a client, we will have a gradual transmission of data in a buffer, that has to be consumed in order to receive more.

3.3.1 Motion JPEG

It is the first attempt of digital video format, where the video signal is encoded as a sequence of frames, and each frame is encoded as a JPG image, meaning it does not take advantage of the clear correlation between one frame and the other.
While encoding video frames, we can omit several data, since there are only a few differences between two images that are close in time, meaning the only thing we store are the differences between images in time. This is done by storing the index of an image, compressing a lot of redundant data, and achieving a lossy compression, which is corroborated by the fact that these index frames are in JPEG format. We hence lose data from chrominance, the JPEG compression (see 53), and in the differences in the two frames. A phenomenon that help this compression is the motion estimation: retrieving from an index image the position (x,y) of an object we can predict the next position of an object starting from an index image, due to the fact that in 35 to 40 frames per second an object cannot move too much. This elaboration is done via a motion vector, but calculating this movement requires **lots** of time, and is done via the Sequential Search algorithm, that is computationally very expensive, going up to $O(p^2N^2)$, and creating hence an *asymmetric encoding*, since this analysis won't be needed in the decoding phase. The high complexity is given by the fact that the algorithm searches for the mean absolute difference, meaning it does the pixel-by-pixel difference divided by two, to the power of two, obtaining a full, bit by bit, search. On smaller images with low resolution we could also use the Hierarchical Search algorithm. The main idea here is to downsample twice an image by a factor of two, going to an image 4 times smaller than the original, then, with the Sequential Source, we find the difference, which, with a cascade effect, gets found in the two bigger images.
The first video standard, developed for videoconferences, is **H.261**, which encodes very small images (352x288), and is also called pstarch, due to the fact it has p channels (up to 30) all sending 64 Kb per second. It develops a symmetric

encoding, since encoding and decoding must happen in real time with a maximum delay of 150 ms. The output is in slow motion, since it varies between 10 and 15 frames per second, while the input is in roughly 30 frames per second. Intra frames are trated as independent images, and use the algorithm of JPEG (not the standard, to avoid the headers), while the inter frames are estimated with motion estimation and compensation. So, again, only differences are encoded, and the content is rebuilt using comparison.
The standard **H.263** instead, uses PB frames for inter frames compression, meaning it is predicted and bidirectional: the difference is hence calculated from the previous and following frame.

3.3.2 Motion Picture Expert Group, MPEG

Once calculated the differences between two frames only a motion vector gets transmitted, which are the direction and movement entity. H.263 allowed the motion vector to refer to pixels outside the boundaries of an image, associating the nearest pixel to the edges of the image, the one pointed by the motion vector, to the outside of the image. This means we can divide an image in a macroblock of 8x8, and for each one of these calculate a motion vector, looking for the most similar macro block in the previous frame. This achieves a better quality with more time consume, which were the reasons the brought to the creation of MPEG.
The first MPEG dates back to 1991, and allows the compression of a sequence of images, stored on a CD, with the aim of random video access and fast research. The compression algorithm is very complex, but most importantly **strongly asymmetric**, since it assures real time decompression. As H.261 standard, MPEG video works in YCbCr color space, with downsampled chrominance components. The sampling used is 4:2:0, and the luminance resolution cannot go higher than 768x576 pixels. It makes also possible to have different resolutions, which allows a frame rate varying from 23.98 to 60 fps. There is also an improvement with respect to H.261 and H.263, since there is spatial redundancy and temporal redundancy: the first encodes a single image with the JPEG encoding, while the second diversifies the encoding for each frame. This is an expansion of the compression algorithm of the previous video standard, since there is a more sophisticated scheme of motion estimation, distinguishing three types of frames:

- I frames (intra coded frame): these frames are encoded with JPEG algorithm, independently and with low quality. Even though, they require higher memory space, but make possible the random access.
- P frames (predictive coded frame): they are encoded based on an estimation referred to the previous I or P frame. The differences are calculated starting from the absolute value of luminance components, occupying a smaller space, with the risk of error propagation.
- B frames (bidirectionally predictive coded frame): two motion estimations are used for encoding, related to the previous and following frames,

in other words, a bidirectional estimation. These are by far the most complex.

This schema is summirized below. More I-frames allow random access in more

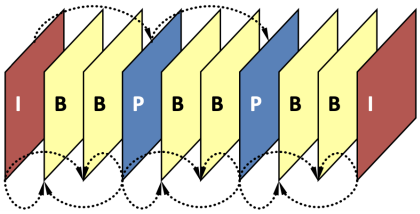


Figure 34: I, B and P frames.

time points, with an exchange of increasing bit-rate. The schema IBBPBBPB-BIBBPBBPBB... is followed, and there must be one I-frame every 15 frames. As for an example, we cannot render the second B frame without the previous I and the following B frames, similarly, without the first we wouldn't be able to encrypt the first P.
The motion compensation and prediction algorithm works in three phases:

1. Motion estimation of objects and motion vector creation.
2. Frames estimation using information collected in the previous phase.
3. Comparison between the estimated frame and the real one to calculate the error.

At the end of this process only the motion vector and the error estimation are saved, this way we can bring with us just the estimation which is way lighter than the actual frame.
MPEG works with a half bit precision, where each 16x16 block is expanded to a virtual copy of 32x32, the search of the new position is done via in the new macro-block, than interpolated in the original one. Again as before, the high complexity comes from the research algorithm. An idea of this algorithm is found in the image below.
One of the main problems of this approach is the size of macro-blocks that need to be applied to the motion compensation prediction algorithm: blocks of bigger size create a lower prediction algorithm, while smaller sizes bring an increasing complexity. Hence, blocks are chosen between variable dimensions, for example, the quad-tree method, where different parts of a frame are decomposed in smaller and smaller blocks in a spiral-like pattern counter clockwise. The advantages of a technique as such is the fact that the predictions are more accurate, meaning also that fewer and fewer differences will need to be encoded. But this is, again, very expensive computationally, and the description of the various *regions* of the macro-block are really complex.
So, as a summary, encode only the difference there are between a frame and the

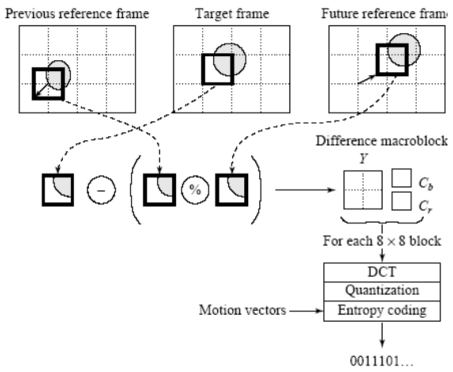


Figure 35: I, B and P frames.

frames before and/or after it, depending on the type of frame we are on (I, B or P). This is done via the motion vector that does a computationally heavy prediction that is compared to the original next frame. After this process only the motion vector (lighter than the actual frame) and the error is saved in the encryption, going to the next frame and so on. This creates a heavy asymmetric encoding, since the encoding phase will be much heavier than the decoding one (the complexity of the motion vector algorithm can be quadratic or more).
MPEG on its own is able to encode CIF images, 352x288, at a comparable quality with respect to the original and a compress ratio of 30:1. It is possible to reach a higher compression ratio, at the price of a lower quality. Its main applications are still today video on CD (demo at most), old videogames and distance education, but not in a real-time streaming scenario.
These results, while not poor, bring to the creation of a MPEG family, which extends to MPEG-1 and MPEG-2, firstly. The first, is able to encode CD-ROM video of medium quality, comparable to the one of a VHS tape. The decoding does not require a specific hardware for standard PCs, so it has a wide spread.

3.3.3 MPEG-2

Differently from its predecessor, MPEG-2 achieves high quality for DVD-ROM, with bitrates higher than 4Mb per second, its closest comparison is an old commercial television broadcasting. It is the standard format for high-quality *consumer applications*, but requires specific hardware for decompression, even though its wide-spread as for today.
MPEG-2 supports 5 different motion prediction procedures, which are: frame prediction for frame-picture, field prediction for field-picture, field prediction for frame-picture, 16x8 MC for field-pictures and dual-prime for P-pictures. Its main differences with MPEG-1 are in the improved error resistance, the support of chromatic under-sampling 4:2:2 and 4:4:4, a non linear quantization and an overall higher flexibility.

3.3.4 MPEG-4

This new advancement in the MPEG family allows to integrate video streams and objects created independently, due to the possibility of indexing single elements of a given scene. It is intended for complex and interactive multimedia systems, but it is supported for different devices and bandwidths, also thanks to the optimization of three different bit-rates (¡ 64Kb per second, 64-384 Kb per second, 384-4Mb per second).

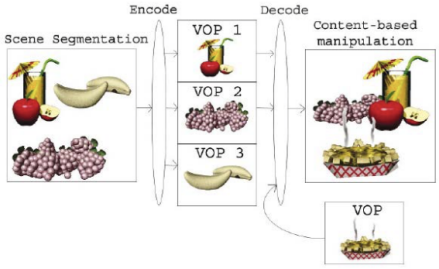


Figure 36: I, B and P frames.

The applications of this protocol are varied and can adapt to modern requirements, like video streaming on the internet, videos for smartphone, virtual meetings, television production and so on.
A video in MPEG-4 can be decomposed in two parts: a background movement, limited to the camera movements, encoded hence as a fixed image, and the actual movements. This creates a hierarchical description of a scene, as such:

1. Video-object Sequence (VS): the complete scene, it contains both natural and synthetic objects.

2. Video Object (VO): a particular scene object, that can have an arbitrary shape, which can be either an object or the background itself of the scene.

3. Video Object Layer (VOL): each VO can have several VOL, or only one, and these components are used for scalable encoding (in the first case) or non scalable encoding.

4. Group of Video Object Plane (GOV): an optional level that allows considering a sequence of VOP.

5. Video Object Plane (VOP): a snapshot of a VO in a particular moment.

The shape of a VOP is arbitrary and must be encoded with the texture, meaning it needs the correct gray-scale. Each VOP is divided in 16x16 blocks, and the motion vector for the global object is calculated starting from this size. Below we can see a VO composition as for MPEG4.
Here we can see how the singular parts of the images were cut, leaving a mask:

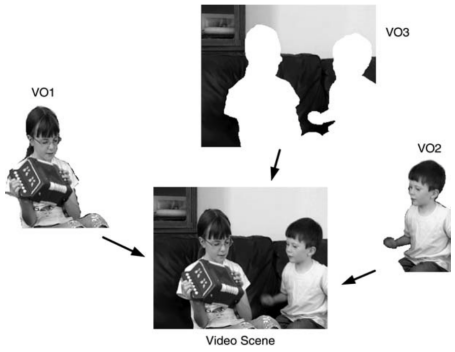


Figure 37: I, B and P frames.

starting from the single element of an image, like the VO1, and adjusting the grey-scale to the borders of it, we could paste it in any different background, composing new frames.

3.3.5 Hints on further MPEG family

MPEG-7 defines how to represent a content descriptor in a standard way, associating to objects a set of descriptors to allow classification and content search. It defines generic containers for objects of different media of different standards, with descriptions thought for human users. It is hence intended for information retrieval. Although, it does not define how to extract content descriptions and how to use those descriptions.
As for the protection of content, MPEG-21 was theorized around 2003, which allows a content description with the rights of who created them. It must provide an interface to make media usage easier.