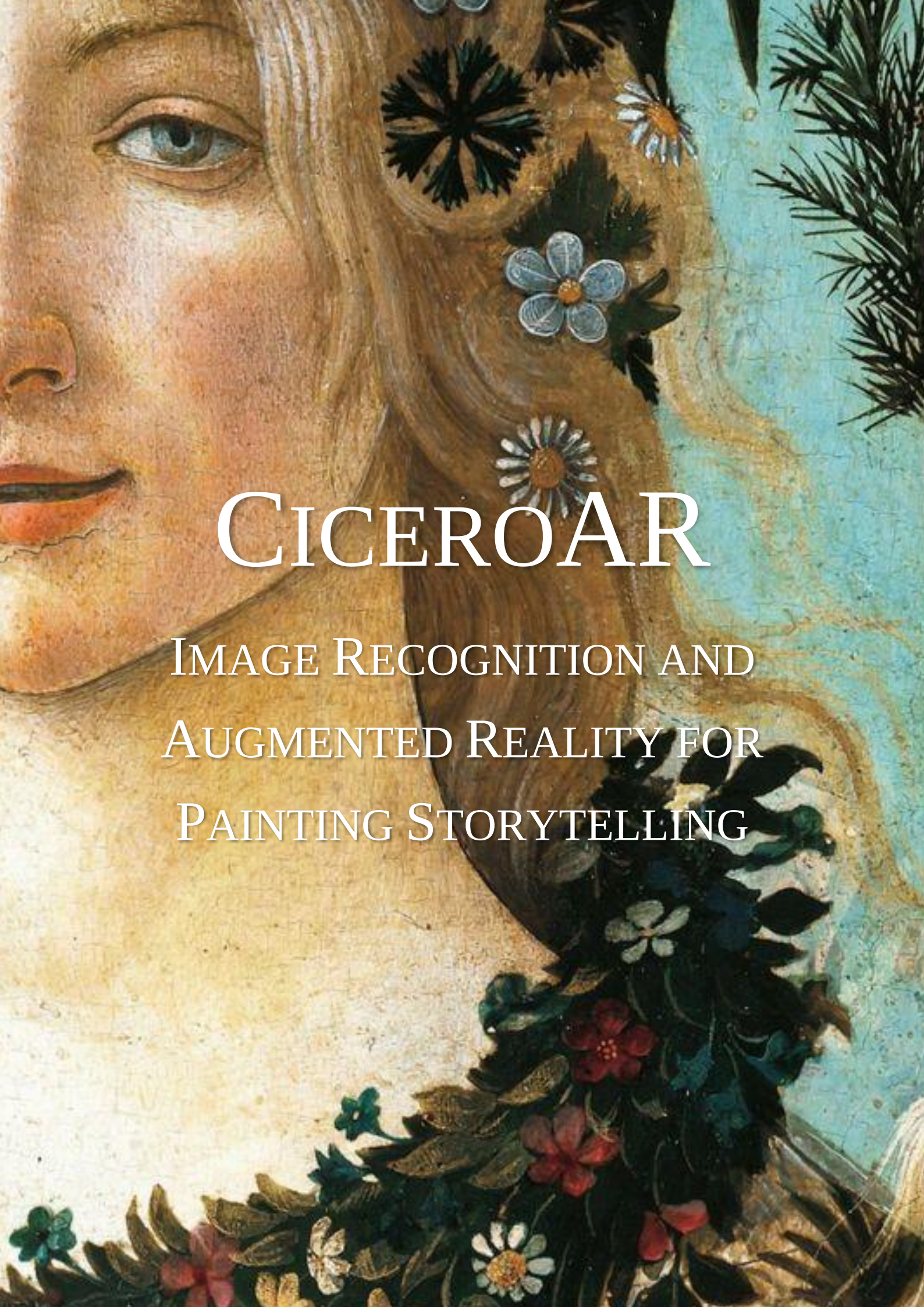


CiceroAR

Image Recognition and Augmented Reality for Painting Storytelling

UNIVERSITÀ DEGLI STUDI DI SALERNO

CORSO DI LAUREA TRIENNALE IN INFORMATICA

CICEROAR:

IMAGE RECOGNITION AND AUGMENTED REALITY FOR PAINTING STORYTELLING

Supervisor: Prof. Andrea F. Abate

Candidate: Michelangelo Esposito – 0512104784

Academic year 2019/2020

INDEX

FIGURE INDEX

INTRODUCTION

When visiting a museum or an art gallery, inspecting a painting is often an activity limited to a quick visual analysis and, in some cases, to a small description found on a plate next to it; for the most part, the history behind a piece and some of its details get lost and forgotten. A guide can help enhancing the discovering experience, however guided tours are not always available, as in the case of small private galleries, or they often provide long routes to which the visitor may not be interested in.

The purpose of this thesis work is to identify alternative support tools for the analysis and the fruition of art pieces, focusing on paintings. The main question that arises is the following: What technologies fit this purpose and are capable of delivering a smart and flexible system that can be used by a wide range of users?

In order to answer this question, we realized a mobile application which magnifies the educational experience provided by a painting, using Artificial Intelligence and Augmented Reality techniques. Through the usage of a convolutional neural network, the implemented system can recognize a piece in a scene and then generate a virtual augmented guide, which will start narrating the painting to the user, while projecting its details in the virtual environment. The application is corrected by a visual interface that facilitates navigation within the different descriptive segments of the work; besides, the employment of Text-To-Speech technology in the narration strengthens the usability of the software for visually impaired or disabled users.

"The Birth of Venus", by Italian renaissance artist Sandro Botticelli was chosen as a case study for the experimentation of the application; such a piece lends itself perfectly to the purpose, given the rich history that characterizes it and the clear spatial separations of characters and elements in the scene depicted.

This thesis is made up of four main chapters: the first two act as the theory base on which our research work, along with the developed application have been founded and built; in the first one, a small overview of Computer Vision is provided, before shifting the focus towards Image Recognition and its realization via Convolutional Neural Networks: the fundamental definitions of an Artificial Neural Network are provided and a few Image Recognition architectures are highlighted. The second chapter is made up of the fundamentals of Augmented Reality: its definition and a brief history are provided, before analysing AR hardware and tracking methods, focusing on the marker-based and markerless approaches.

The final two chapters are dedicated to the application we developed during an internship at the I.T. Svil Ltd. in Salerno across the span of a few months, CiceroAR; in most instances we use UML diagrams in order to provide information in a schematic way, accompanying them with a description, when necessary.

The third chapter contains the general principles of the system, focusing on a high abstraction level: after a general introduction, accompanied with an activity diagram depicting the expected flow of events, the goals and requirements of the system are identified, followed by a series of user-centred scenarios, detailing the interaction with the application in different contexts and by different users. Finally, a more formal use case model is provided, in which the scenarios are used to generalize the system's usage.

In the fourth and final chapter, we discuss the details of the application from an architectural and software point of view; compared to the previous one, this chapter is a lot more technical, since it exposes the underlying logic of the system. After a general architecture overview, a series of code snippets are provided; these have been extracted from the source code of the application, in order to detail and explain the main functionalities of the system from a lower abstraction point of view. The different choices we encountered during the development, as well as the tools and technologies we used are also extensively discussed and confronted among themselves.

In the last paragraph, we provide an overview of the final application, accompanying it with a series of screenshots depicting the running software. Furthermore, we decompose the painting use case on which the testing of the system was based, by presenting insights on the training process and the analysis of the painting itself.

1. *IMAGE RECOGNITION*

As humans, the perception of what surrounds us happens daily and with relative ease: recognizing the physical properties of our world, such as the colour of an object, the texture of a surface or the translucency of a panel, quickly becomes an unchallenging activity; an indisputably harder ability, like deriving context information from an environment, is still a capacity we develop early on in our life.

Computer Vision (CV) is a scientific field that deals with how computers can gain high-level understanding from digital images or videos; in the last decades, impressive milestones have been reached in various fields: in medicine, segmentation of brain tumor has high clinical relevance for the estimation of the volume and spread of a tumor and skeleton segmentation techniques have been able to provide a fast and reliable 3D observation of fractured bones; in the security industry, CV techniques such as real-time face recognition or object detection, combined with biometry are able to provide an easier control over entire buildings.

<<However, despite all of these advances, the dream of having a computer interpret an image at the same level as a two-year old remains elusive.>> [1]

So, why must an "intelligent" machine resort to physics-based and probabilistic models to disambiguate between different possible solutions, when describing the world and reconstructing its properties is such an effortless task for humans?

The problem is based both on the still limited understanding of biological vision and on the complexity of vision perception in a dynamic and nearly infinite varying physical world; focusing on the latter from a machine perspective, the recognition problem can be broken down into several components: if we know what we are looking for, the problem falls into object detection, which involves quickly scanning an image to determine where a match may occur. If we have a specific rigid object we are trying to recognize, we can search for characteristics feature points and verify that they align in a geometrically plausible way. A more challenging version of recognition is general category recognition, which may involve recognizing instances of extremely varied classes such as animals or furniture. Some techniques rely purely on the presence of features and their relative positions, while others involve segmenting the image into semantically meaningful regions; regardless, in most instances, recognition depends heavily on the context of surrounding objects and scene elements.

1.1 IMAGE RECOGNITION USING CONVOLUTIONAL NEURAL NETWORKS

Image Recognition is a computer vision technique that allows machines to interpret and categorize what they "see" in images or videos. While recognizing image patterns and extracting features is often the initial step of more complex computer vision techniques, like object detection or image segmentation, there are various standalone applications that make the technique an essential Machine Learning task and the employment of neural networks has become the state-of-the-art approach for it. With this technology, a Machine Learning model is trained to receive an image as the input and output a target class, which is a label or a set of labels describing the image; usually the model also outputs a confidence score along with the predicted class: this is nothing more than the probability that the image belongs to that class, according to the model. The technique can be broken into two separate branches: single and multiclass recognition; in single class image recognition, a model, or binary classifier, predicts only one label per image. On the other hand, multiclass models can assign several labels to a single image, outputting a confidence score for each one.

Nearly all image recognition models begin with an encoder (or network), which is made up of blocks of layers that learn statistical patterns in the pixels of images that correspond to the labels they are attempting to predict; this encoder is then linked to a fully connected layer, which outputs the confidence scores for each label.

Generally speaking, an Artificial Neural Network (ANN) is an algorithm designed to recognize patterns in data and group them together; it is based on a collection of connected units or nodes, called artificial neurons, which can receive signals, process them and then signal the other neurons connected to them through connections called edges. Each neuron and edge can have an adjustable weight, which increases or decreases the strength of the signal received. The original goal of the ANN approach was to solve problems in the same way that a human brain would, however over time the focus shifted to performing specific tasks in various fields, such as computer vision, speech recognition, machine translation or medical diagnosis.

Formally, A neural network is a sorted triple (N, V, w) with two sets N, V and a function w, where N is the set of neurons and V a set $\{(i, j) \mid i, j \in N\}$ whose elements are called connections between neuron i and neuron j. The function $w: V \rightarrow R$ defines the weights, where $w((i, j))$, the weight of the connection between neuron i and neuron j, is shortened

to $w_{i,j}$. Depending on the point of view it is either undefined or 0 for connections that do not exist in the network.[2]

Looking at a neuron j and at its connected neighbours, the propagation function of j is defined as the function used to transport values through neuron j; it usually receives the outputs $o_{i1}, \ldots, o_{in}$ of other neurons $i_1, ..., i_n$ and transforms them in consideration of the connecting weights $w_{i,j}$ into the network input net_j that can be further processed by the activation function. The network input is thus the result of the propagation function.

Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of neurons, such that $\forall z \in \{1, \ldots, n\} : \exists w_{iz,\ j}$. Then the network input of j, called net_j is calculated by the propagation function f_{prop} as follows:

$$\text{net}_j = f_{prop}\ (o_{i1}, \ldots, o_{in}, w_{i1,\ j}, \ldots, w_{in,\ j}) = \sum_{i \in I} (o_i \ .\ w_{i,j})$$

The activation state of a neuron indicates the extent of a neuron's activation and is often referred as just activation. Its formal definition is as follows:

Let j be a neuron. The activation state a_j is explicitly assigned to j, indicates the extent of the neuron's activity and results from the activation function. Each neuron may also have a threshold such that a signal is sent only if the aggregate signal crosses the threshold: This value, Θ_j is uniquely assigned to j and marks the position of the maximum gradient value of the activation function.

Let j be a neuron. The activation function is defined as:

$$a_j(t) = f_{act}\ (\text{net}_j(t), a_j(t-1), \Theta_j)$$

It transforms the network input net_j, as well as the previous activation state $a_j(t-1)$ into a new activation state $a_j(t)$, with the threshold value Θ playing an important role, as already mentioned. The activation function is usually defined globally, for all neurons, and only the threshold values are different for each neuron and can be adjusted by a learning procedure.

When it comes to networks architecture, different topologies exist for ANNs: in a feedforward network, the neurons are clearly separated into one input layer, one or more hidden processing layers and one output layer. Neurons in one layer have only directed connections to the neurons of the next layer, towards the output layer; if each neuron is connected to all neurons of the next layer, the topology is completely linked. Some feedforward networks allow "shortcut connections": connections that skip one or more levels and are usually directed towards the output layer.

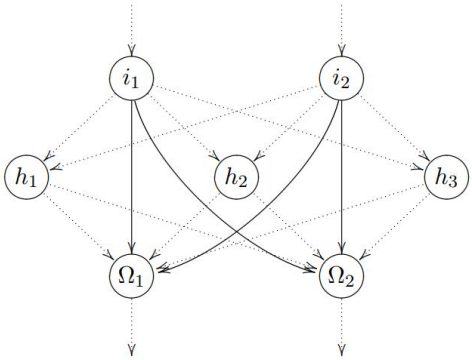


Figure 1.1 - A feedforward network with shortcuts

Some networks allow the neurons to be connected to themselves (direct recurrence) or to the input layer (indirect recurrence): in the first case neurons inhibit and strengthen themselves in order to reach their activation limits; with indirect recurrence, on the other hand, a neuron can indirect forwards connection in order to influence itself.

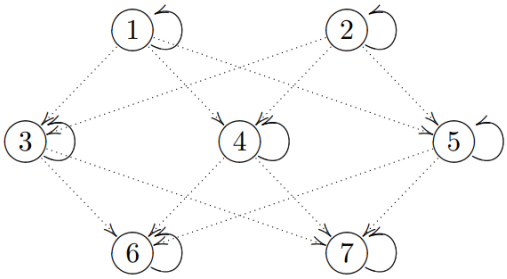


Figure 1.2 - A network containing recurrent nodes

Finally, a completely linked network permits connections between all neuron, except for direct recurrences; furthermore, the connections must be symmetric.

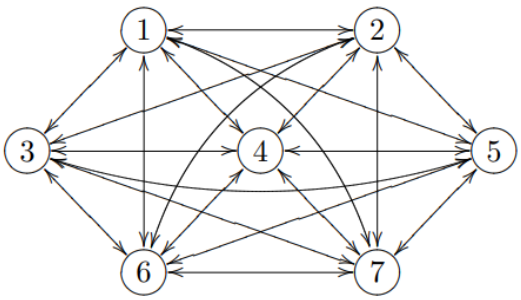


Figure 1.3 - A completely linked neural network

Suitable architectures for Image Recognition are based on variations of Convolutional Neural Networks (CNNs). A CNN is an algorithm which can take an input image, assign importance to various aspects/objects in it (learnable weights and biases) and be able to differentiate one from the other. While in primitive methods filters are hand-engineered, with enough training, CNNs have the ability to learn these filters/characteristics.

The architecture of such a network is analogous to the connectivity patterns of human neurons and was inspired by the organization of the visual cortex, where individual neurons respond to stimuli only in a restricted region of the visual field, known as receptive field; a collection of such fields overlap to cover the entire visual area.

The first part of the CNN consists of convolutional and max-pooling features extractor layers, while the second part consists of the fully connected layer which performs non-linear transformations of the extracted features and acts as the classifier. If the neurons in the convolutional layer find the features they are looking for they produce a high activation.

In image processing, to calculate convolution at a particular location (x, y), a k x k sized chunk, the kernel, is extracted from the image, centred at location (x ,y); the values in this chunk are then multiplied element-by-element with the convolution filter, also sized k x k, and then they are added together to obtain a single output.

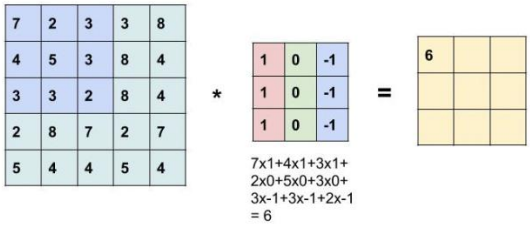


Figure 1.4 - Convolution operation example

Storing an image means keeping track of the colour information associated to each individual pixel in a colour matrix; the size of each pixel depends on the colour depth (8-16-24 bit). Once images reach a notable dimension, calculations can get very intensive, so the role of the CNN is to spatially reduce the images into a form easier to process, without losing features which are critical for prediction sake.

This is achieved by a max pooling layer, which is responsible of reducing the spatial size of the image (not its depth); this reduces the number of parameters, avoiding overfitting, the condition when a trained model learns too much out of the training data and loses the ability to generalize. A common form of pooling is max pooling where a

filter of size p is taken and the maximum operation over the sized part of the image is applied.

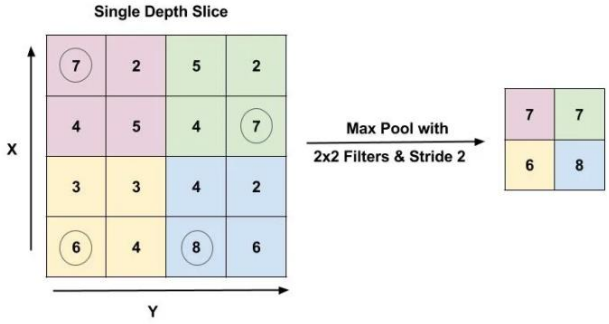


Figure 1.5 - Max pooling operation example

The fully connected layer is made up of an ANN, which purpose is to combine the detected features into more attributes, in order to predict the classes with greater accuracy.

Many neural network architectures [9] exist for image recognition, including:

- AlexNet: deep neural network winner of the ImageNet classification in 2012; it's widely credited with sparking a resurgence of interest in using deep convolutional neural networks to solve computer vision problems. The network is relatively large, with over 60 million parameters and many internal connections, thanks to dense layers that make the network quite slow to run in practice.
- VGGNet: network developed by researchers from the Visual Geometry Group (VGG) at Oxford University. VGGNet has more convolution blocks than AlexNet, making it "deeper", and it comes in 16- and 19-layer varieties, referred to as VGG16 and VGG19, respectively.

1.2 DATA SCIENCE PROCESS OVERVIEW

Image Recognition and Machine Learning in general are both task that heavily rely on data to use for their training process, in order to disambiguate between different possible positive candidates. The following diagram [8] highlights the essentials step of a data science process, starting from the setup of the research goal, all the way to the presentation and automation of the model.

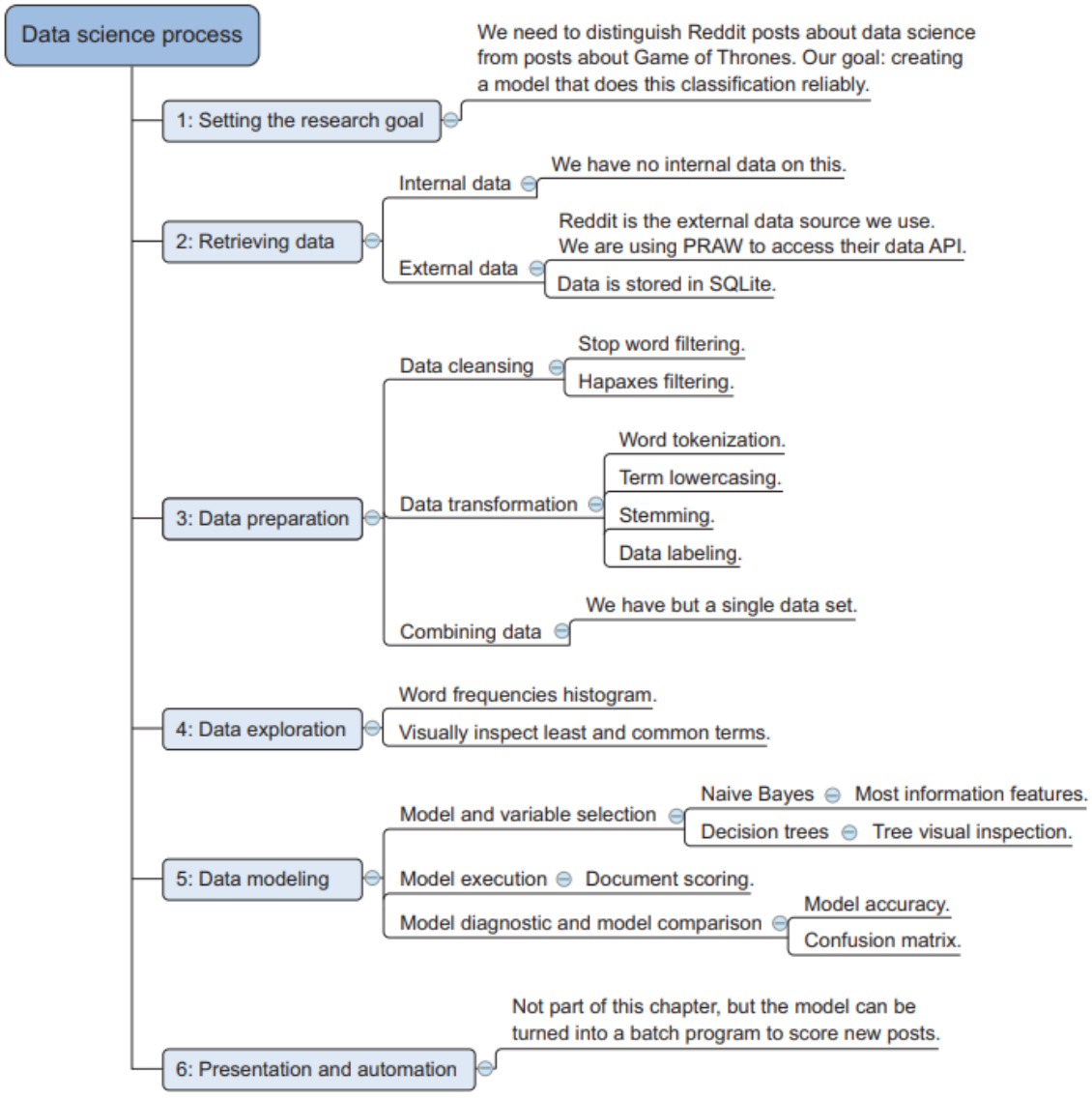


Figure 1.6 - Data Science process for an example application

The first step of any data science project is to determine its goal and evaluate it the technology is suitable for the purpose. Once the goal is set, the data source must be identified: this can be internal, such as user generated data, or external, where the data is fetched from an external source (free online databases, websites, …).

Before performing any analysis, some operations on the retrieved data may be performed, since oftentimes data can be quite messy, especially if it wasn't generated with any particular data science purpose in mind or it wasn't well maintained during its lifecycle; if the data is textual common operations to perform are word filtering, term lowercasing or labelling; if on the other hand, we are working with images, these are usually rescaled, colour-transformed or labelled. Data preparation is the most crucial step required to get correct results. The fourth step is data exploration: once the data is filtered, rescaled or labelled, its sheer size can hinder us from getting a good grip on whether it's clean enough for actual use; in this step further transformations may be applied. During data modelling, statistical, mathematical and technological knowledge is applied to the data in order to find any possible insight: comparison algorithms or Machine Learning models are executed on the data and the result is then tested, often with unseen data, and its performance is measured. Finally, the results of the process can either be turned into a useful application or presented to others using various data visualization techniques.

2. AUGMENTED REALITY

2.1 AR OVERVIEW

Augmented Reality (AR) is a relatively new technology which is focused on the blending of digital elements, such as visual overlays or 3D models and animations, into real-world environments: with the help of advanced AR technologies (e.g. adding computer vision, incorporating AR cameras into smartphone applications and object recognition) the information about the surrounding real world of the user becomes interactive and digitally manipulated. Furthermore, an AR experience can enhance the overall user experience by providing additional stimuli for their other senses, in addition to visual augmentation: a system can improve the immersivity of an application with augmented sounds, scents and haptic feedbacks.

<<Achieving this connection is an incredible goal, one that draws upon knowledge from many areas of computer science yet can lead to misconceptions about what AR really is. For example, many people associate the visual combination of virtual and real elements with the special effects in movies. While the computer graphics techniques used in movies may be applicable to AR as well, movies lack one crucial aspect of AR, interactivity.>> [3]

The most widely accepted formal definition of AR was proposed by R. Azuma in his 1997 paper, "A Survey of Augmented Reality". According to him, AR must have the following three characteristics:

- Combines real and virtual;
- Interactive in real time;
- Registered in 3D.

A complete AR system requires at least three components: a tracking component, a registration component and a visualization component; a fourth component, a spatial model, may be used to store information about the real world and about the virtual world: the real world information is required in order to provide a reference for the tracking component, which is tasked with determining the user's location in the real world, while the virtual world model handles the content used for the augmentations; they both must be registered in the same coordinate system. What essentially happens during an AR interaction is a feedback loop between the user and the system: the user observes their device's screen and therefore controls the viewpoint, while the system

tracks their movements and registers the pose in the real world, with the virtual content, before presenting it by drawing the virtual image on top of the camera image.

Compared to Virtual Reality, where the user's perception of reality is completely based on virtual information, Augmented Reality is considered an example of "Mixed Reality", since it incorporates both real world and fictional elements. For example, in architecture, VR can be used to create a walk-through simulation of the inside of a new building, while AR can be employed to show a building's structures and systems superimposed on a real-life view.

The term "Mixed Reality" was introduced by Paul Milgram in a 1994 publication called "Augmented Reality: A class of displays on the reality-virtuality continuum"; this paper presented the idea that between virtual and real there is a spectrum of different mixtures of both.

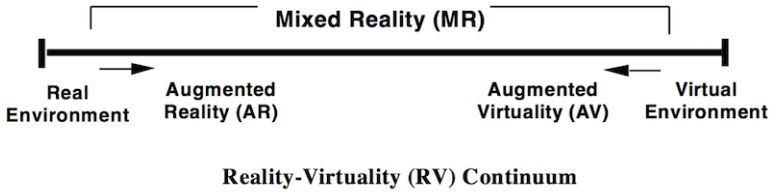


Figure 2.1 - Milgram et al., The Reality-Virtuality Continuum

The term Mixed Reality was introduced as follows:

<<Within this framework it is straightforward to define a generic Mixed Reality (MR) environment as one in which real world and virtual world objects are presented together within a single display, that is, anywhere between the extrema of the RV continuum.>> [5]

The first "AR experience" was achieved, to some extent, by a cinematographer called Morton Heilig in 1957: he invented the Sensorama, a machine capable to deliver visuals, sounds, vibrations and smell to the viewer; obviously it wasn't computer controlled, however it was the first example of an attempt at adding more depth to a static visual experience. The first actual AR system ever built was developed by Louis Rosenberg in 1992 at the USAF Armstrong Labs; it was called Virtual Fixtures and was a complex robotic system, designed to compensate for the lack of high-speed 3D graphics processing power in the early 90s. In order to create the augmented experience, a unique optics configuration was employed that involved a pair of binocular magnifiers aligned so that the user's view of the robot arms was brought forward so as to appear registered in the same location of the user's real arms.

Today there are applications available for or being researched for AR in nearly every field, including archaeology, art, medicine, military industry and entertainment; it is expected that even more potential areas of application are soon to be rising. The technology is well suited for on-site visualization both indoors and outdoors, for visual guidance in assembly, maintenance and training; it enables interactive games, social applications and new forms of advertising: in interior design, AR allows users to virtually preview a piece of furniture and test its look in their own living rooms; in assembly, an AR system can guide the assembler through each step, visualizing the relative instructions or its interaction points; finally, in the game industry, where accuracy is less critical than in the medical or military industry, AR can make games more attractive by providing new mechanisms of interaction between the player and the game. Although some impressive milestones and goals have been reached in recent years, due to the technology still being relatively young, a lot of areas still require further research before the employment of highly reliable AR systems, especially in fields where the precision of the tracking systems is crucial.

2.2 AR HARDWARE AND TRACKING METHODS

<<In Augmented Reality, virtual objects supplement rather than supplant the real world. Preserving the illusion that the two coexist requires proper alignment and registration of the virtual objects to the real world. Even tiny errors in registration are easily detectable by the human visual system.>> [6]

This contributes to the identification of a key measurement for AR systems: how realistically they integrate augmentations with the real world: such a system must be able to derive real world coordinates, independently of the camera used to move through the scene, and interpret the semantic context of the environment; all of this must happen in real time in order to allow interaction.

Traditional motion tracking devices can be classified based on the technologies used for the measurements: mechanicals, electromagnetic, optic, acoustic and inertial; these can be further categorized based on a series of reference parameters:

- Work volume: the physical region of space in which the device works reliably;
- Sampling rate: the rate at which the device detects and updates variables in the environment;
- Resolution: the smallest position variation detected.
- Latency: the time slot between an event and its recognition;
- Precision of the system.

Mechanical tracking systems are arm-based systems that use potentiometers or optic encoders to measure the rotation of the connecting pins of the connecting rods. Once the angles of each joint and the length of the chain rods kinematics are known it's possible to easily calculate the position of the object to track. These systems are relatively cheap and completely latency-free, due to the lack of a transmission-reception component; however, their work volume is usually small, and the movable parts are subject to usury.
Electromagnetic systems are based on the usage of a transmitter and a receiver: the transmitter generates a floating magnetic field by means of three orthogonal spires; this field is detected by three similar spires in the receiver component and the variations in the strength of the signal is interpreted as a variation of the position of the object which is being tracked. These devices are relatively small and can be easily installed on the user's body or on other small objects and, although the work volume isn't quite big, it's possible to increase it by chaining different devices together. Interferences from other electronic devices are the main disadvantage of using such a system.
Optical tracking systems are highly accurate, however they present high complexity and costs as well; they work by using a series of light sources (usually infrared), placed on the object of interest and employing cameras to detect their movements.
In acoustic-based tracking an emitter generates an audio signal which is picked up by a microphone, measuring the time necessary to the sound to traverse the path. Data from three devices is processed and used to calculate the position and the orientation of the object. The technology is cheap, however the speed of sound variates with weather conditions such as pression and temperature, making the system unreliable.
Inertial systems use gyroscopes to measure changes in the rotation of the object along one or multiple axes.
Later, the concept of user movement tracking drastically changes, as researchers in various fields, such as computer vision and robotics, have developed a series of different tracking techniques and algorithms, which can be classified, based on the equipment used, into visual tracking methods and hybrid methods. In visual tracking, the system deduces the camera pose based on observations of the scene; this is considerably hard in an unknown environment since it requires effort to collect enough data to detect the pose, which will still fluctuate over time. A simple solution to overcome this obstacle is to add an easily recognizable element in the environment. This element is called marker and is a sign or an image built in such a way to be recognized by a computer system, via

image processing, pattern recognition and computer vision techniques, and then used as a reference point in a scene.

Other approaches for visual tracking are feature-based and model-based methods; in model-based tracking, the system has a model of the scene or part of the scene and what happens is essentially a comparison between the visual observations and the model, from which a best match is found to define the pose of the camera, while in feature-based tracking the system detects optical features in the images and learns the environment based on observations of movements between frames.

In marker-based AR applications, in order to be easily and reliably detectable under all circumstances, a marker must follow specific constraints:

- It must be perfectly squared, with well-defined proportions;
- The external borders must be well defined and continuous;
- The inner image should be asymmetric, so there are some aspects of the marker that make it possible for the vision software to determine which way the marker is oriented:
- Different markers must be as loosely coupled as possible; this way, when the camera is far away from the image, which is represented by fewer and fewer pixels the greater the distance, it is still possible to distinguish it from other markers.

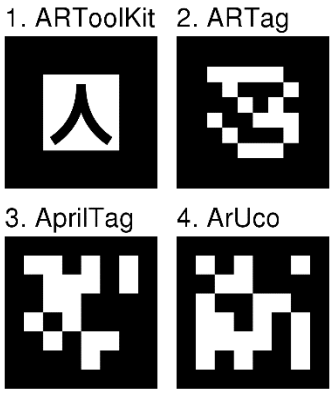


Figure 2.2 - Comparison of some Augmented Reality markers.

In “Theory and applications of marker-based Augmented Reality” [7], S. Siltanen describes the basic marker detection procedure as a series of steps:

0. Image acquisition
    - acquisition of an intensity image.

1. Pre-processing
    - low level image processing
    - undistortion
    - line detection/line fitting

2. Detection of potential markers and discard of obvious non-markers
    - fast rejection of obvious non-markers
    - fast acceptance test for potential markers

3. Detection and decoding of markers
    - template matching (template markers)
    - decoding (data markers)

4. Calculation of the marker pose
    - estimation of the marker’s pose
    - iterative pose calculation for accurate pose

The image acquisition step is actually a separate process and just provides the image for the marker detection procedure.
Before the actual detection of the marker, the system must obtain an intensity image (a grayscale image). If the capture image format is something else, the system converts it. The first task of the marker detection process is to find the boundaries of the potential marker. Two approaches are used by detection systems to achieve this: either they first threshold an image and search for markers from the binary image, or they detect edges from a grayscale image. In traditional offline computer vision applications, the captured image is undistorted using the inverse distortion function calculated during the camera calibration process; in AR, systems typically undistort only the locations of feature points in order to speed up the system. Applications use several methods for line detection, line fitting and line sorting: methods based on edge sorting are generally

robust against partial occlusion, but their computational cost is more expensive, which makes them unsuitable for mobile devices.

Since time is of the essence in AR applications, many implementations use fast calculable acceptance/rejection criteria to distinguish real markers from objects that are clearly something else. First, the histogram of a black and white marker is bipolar, and the marker detection system may check the bipolarity as a fast acceptance/rejection criterion, however eventual reflection may create grey scale values, which are to be considered. Calculating the number of pixels belonging to the perimeter is a very fast process, and a system can carry it out as part of the labelling process, whereas calculating the exact size of the object is more complicated and time consuming. Therefore, it is rational to estimate the size of an object using the number of edge pixels, for example. Another useful technique used when the system has information about the overall appearance of the marker, which contain a small known number of holes (white areas) is to calculate the number of said holes and use it as an acceptance/rejection criterion during pre-processing.

The pose of an object refers to its location and orientation: the location can be expressed with three translation coordinates, (x, y, z), while the orientation can be represented as three rotation angles, (α, β, γ). Since the pose of a calibrated camera can be uniquely determined from a minimum of four coplanar but non-collinear points, a system can calculate a marker's pose, relative to the camera using the four corner points of the marker in image coordinates. A few complications take place during the pose calculation process: the detection of x and y translations is more reliable than the detection of z translation. The camera geometry explains this. If an object moves a certain distance in the z direction, the corresponding movement on the image plane is much smaller that if it moves the same distance in the x or y direction. Vice versa: small detection errors on the image plane have a greater effect on the z component of the translation vector than on x and y components. In addition, the pose of a marker seen from the front is more uncertain than a pose of a marker seen from an oblique angle.

By using the marker-based approach, tracking becomes possible as long as the marker is in the camera's field of view; additionally, any eventual noise in the camera stream can impact on the detection of the only marker in the scene. Some workarounds exist for these limitations such as the multi-marker approach, where a cluster of markers is used to allow for a freer camera movement in the scene. With this method, the detection

is more fault tolerant, since a detection error on one of the markers, caused by camera noise or lighting conditions, can be corrected by using the other markers.

Since the usage of markers is not always possible due to physical limitations, and with the recent emergence of advanced camera systems and more precise sensors in mainstream devices, markerless AR solutions have become an industry preference.
The technique requires little to no environment preparation, however the tracking itself becomes much more complicated; it uses a combination of camera systems, dedicated sensors and complex math to effectively detect and map the real-world environment; after obtaining a map of the area, the AR application makes it possible to place and track the virtual objects without the need of any marker in the scene. With this approach, tracking works great in most of the cases, but becomes a problem when the target surface is not easily identifiable or has no distinctive texture, since not many features can be extracted from it.
Markerless AR has recently seen its biggest impact in the gaming industry, with the AR-enabled Pokémon Go becoming a git in 2016, however fields such as advertising and education have seen a rise in similar AR-focused applications in the recent years as well.
Both technologies show advantages and limitations; for this reason, hybrid tracking techniques also exist: the combined usage of a multi-marker system and hardware such as gyroscopes and accelerometers which calculate and predict the user's movements, allows to compensate for tracking errors when detection fails.

3. APPLICATION OVERVIEW

In this chapter we will provide the initial overview of the application we developed for this work of thesis, CiceroAR; the software is the result of an internship at the I.T. Svil Ltd. in Salerno, across the span of a few months.

CiceroAR aims to deliver an enhanced educational experience through the usage of Augmented Reality and Artificial Intelligence technology, in order to provide the user with additional information about a painting or help visually impaired people receive auditive aid via Text-To-Speech tools. Anyone with a supported device can come across new ways to appreciate art and even discover new details in an evocative piece, simply by using their smartphone's camera and letting the system do the work behind the scenes.

The following UML activity diagram provides an initial high-level understanding of the system's typical flow of events by depicting the sequence of activities performed by a user in CiceroAR.

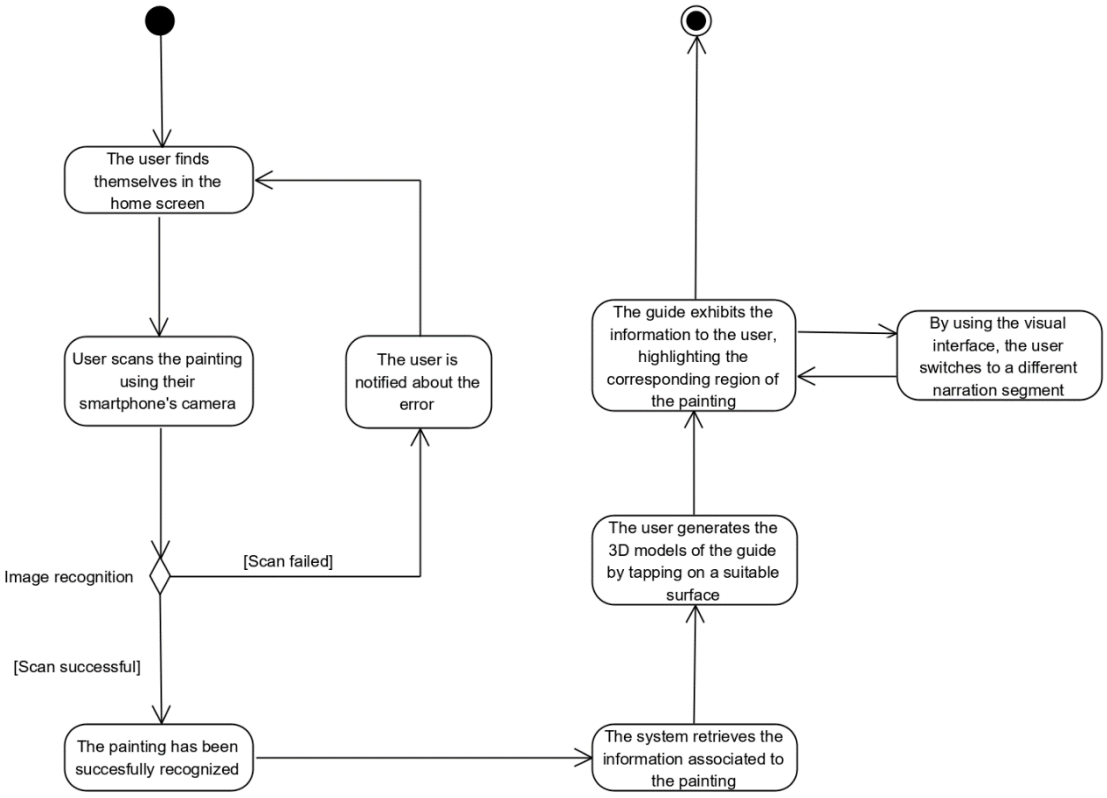


Figure 3.1 - UML activity diagram depicting system's workflow.

After launching the application and getting past the home screen menu, a user has the ability to open their smartphone's integrated camera and use it to take a picture of a painting in a scene; the system then begins a step in which it processes the captured image, trying to find one of the supported paintings in it. If the image is recognized, the system looks up and retrieve its associated information, manually prepared beforehand. At this point the user can generate a virtual augmented guide in the scene, by tapping on a suitable surface on their device's screen. Once placed, the guide begins interacting with the user via Text-To-Speech technology and starts narrating the previously retrieved information; while doing so, the corresponding regions of the painting are highlighted by being projected forward in the augmented scene, when possible. At any time during the storytelling process, the user can interact with the narration by pausing or resuming the guide's voice; additionally, by using the application's interface, it is possible to navigate to the previous and next storytelling segments, corresponding to different sections of the painting.

3.1 GOALS AND REQUIREMENTS

In this section we present and highlight CiceroAR's goals and system requirements, both functional and non-functional.

These system goals will provide a solid base on which to build the application:

- Enhance the educational experience of browsing art for students and art enthusiasts;
- Create a more interactive alternative to traditional museum guides;
- Aid visually impaired people experience art in a new way.

Functional requirements are descriptions of the service that the system must offer; they help checking whether the application is providing all the mentioned functionalities and can help to catch initial design error, which are much cheaper to fix at this stage of the development. The identified functional requirements are:

- **FR_01**: The system must allow the user to scan a painting with the smartphone's integrated camera;
- **FR_02**: The system must be able to identify a painting in an image;
- **FR_03**: The system must be able to retrieve the information associated to a scanned painting;
- **FR_04**: The system must be able to project an augmented reality guide in the environment;

- **FR_05**: The system must be able to highlight the details on the painting by using Augmented Reality;
- **FR_06**: The system must utilize Text-To-Speech technology when providing the user with the requested information;
- **FR_07**: The system must allow the user to pause and resume the storytelling;
- **FR_08**: The system must allow the user to skip ahead or backtrack to the next and previous narration segments during the storytelling;
- **FR_09**: The system must allow an administrator to add, update or remove a painting and its related information to/from the archive.

Non-functional requirements, on the other hand, are requirements that specify criteria that can be used to objectively describe the system when performing a specific task or operation; they are architecturally significant requirements and they dictate the development during the system design phase. We identified a small set of non-functional requirements (here, U stands for Usability, R for Reliability, P for Performance and I for Implementation):

- **NFR_U01**: The system's interface must be easy to use and not ambiguous;
- **NFR_U02**: The system must ensure operations are performed in the most direct way available;
- **NFR_R01**: The information provided by the system must always be reliable and consistent with the referenced painting;
- **NFR_R02**: The system must ensure any error message is delivered to the user in less than 5 seconds;
- **NFR_P01**: System response time must not exceed 5s when performing recognition operations;
- **NFR_P02**: The system must support concurrent user requests;
- **NFR_I01**: The system must run on any Android device released in the last 2 years, with an API version of 28 or higher (Android Pie), or any x86 or x86_64 AVD;
- **NFR_I02**: The system's client side must be implemented in Java, as a native Android application.

In order to illustrate the usage of the application at a high abstraction level, a series of scenarios have been crafted; these highlight the sequence of action performed by different kinds of users in different environments, when the system is operational.

Each scenario is characterized by a name, a unique identifier, its participants and the flow of actions that make up the usage example.

This first scenario depicts the most expected use of the application: scanning a painting when visiting a museum.

Scenario name	MuseumUsage
Scenario ID	SC_01
Participants	Ann: art student visiting the Salvador Dalí Museum in Figueres with her class.
Flow of events	1. Ann is wandering into the halls of the museum when a particular piece catches her eye, Palladio's Corridor of Thalia, so she gets closer and starts to examine it. 2. The girl isn't satisfied with the little information provided by the plate next to the painting and wants to know more. 3. Ellie, one of Ann's classmates suggests her CiceroAR, to quench her thirst for more details. 4. Ann decides to give it a try, so she downloads the app and, after reading the manual, tries to scan the painting using her smartphone's camera to take a picture of it. 5. The system recognizes the painting and proceeds to generate an interactive guide in the space in front of Ann, using AR. 6. The guide then begins to narrate the story of the painting to Ann, in particular it tells Ann how the piece is heavily influenced by Italian Renaissance art. 7. Moving to the painting itself, the guide starts to speak about the strongly lit figure of a girl playing with a skipping rope in the top left, while the corresponding region is projected towards her in the virtual environment.

This second scenario aims to provide an example usage of the application in a home environment, by using a book as the source of the painting to recognize; the application behaves correctly and identifies the image.

Scenario name	HomeUsage
Scenario ID	SC_02
Participants	Frank: art enthusiast working from home during quarantine.
Flow of events	1. After being stuck at home for more than three weeks, Frank decides to go through his old photo books. 2. While browsing the pages, the man notices a picture of him next to “The Bedroom” by Vincent Van Gogh, snapped during a trip some years ago. 3. Frank decides to download CiceroAR and try to use it to get more info on the painting in the picture, so using his phone he takes a picture of the page. 4. The application then scans the captured image looking for a painting. 5. The system correctly recognizes it and proceeds to generate the guide, which then begins the narration process.

This last scenario highlights a use case that, while not as frequent as the others, is just as important for the accessibility of the application: the usage of the system by a visually impaired person.

Scenario name	AidedUsage
Scenario ID	SC_03
Participants	Ann: visually impaired art lover.
Flow of events	1. Ann is visiting the Uffizi museum in Florence and is making her way through the halls. 2. Once she stumbles upon “The Birth of Venus” by Sandro Botticelli, she is not satisfied by the plaque sitting next to it, since she finds it really hard to read it. 3. Ann decides to use CiceroAR as an alternative, so after taking a picture of the painting and letting the system process it, she generates the augmented guide which proceeds to narrate the details to the woman, without the need to read anything.

3.2 USE CASE MODEL

From the identified scenarios a more formal use case model has been constructed: this provides a series of use cases performed by the different actors of the system, along with the flow of events which describes their execution, including eventual exceptions or unexpected behaviours. Two actors have been identified during this phase: a generic user of the system and an administrator; the first has access to the main functionalities of the application, like the ability to scan a painting, generate the augmented guide and listen to the narration , while the latter is in control of the system's data, with possibility to add, remove or update a new painting and its information. Further boundary use cases will be later discussed, after introducing the system's architecture.

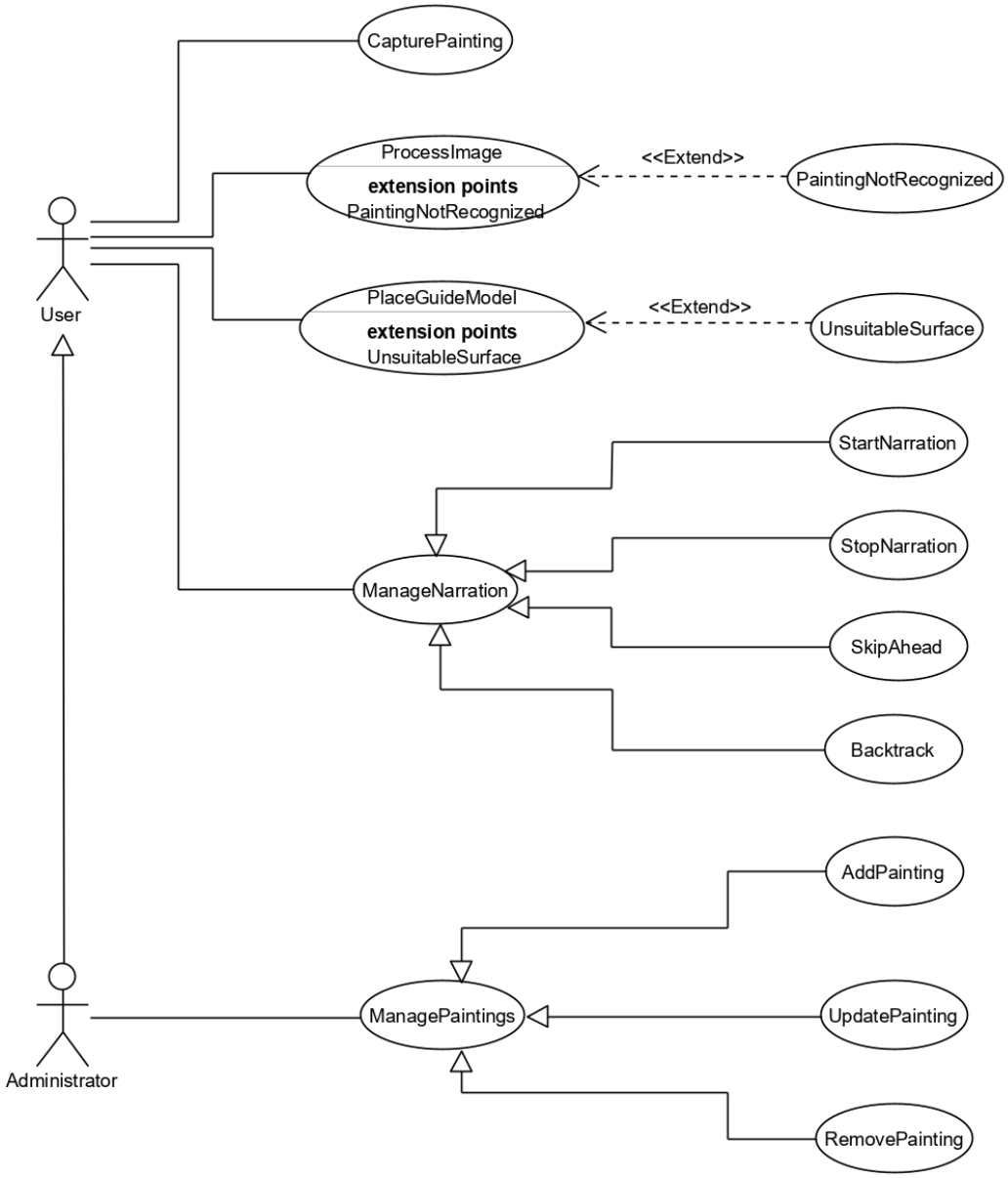


Figure 3.2 - UML Use Case Diagram of the system

Use case name	CapturePainting
Use case ID	UC_01
Participating actors	User
Preconditions	The user finds themselves in the home screen
Flow of events	1. The user opens the device camera, points it towards the painting and captures the image. 2. The system displays a preview of the captured image to the user.
Postconditions	The image was successfully captured, and the user is looking at its preview.

Use case name	ProcessImage
Use case ID	UC_02
Participating actors	User
Preconditions	The user is sitting in the image preview page.
Flow of events	1. By using the interface, the user starts up the image processing. 2. The system processes the image and displays a message to the user while doing so.
Postconditions	The image was successfully processed, and the user is notified.
Exceptions	The system did not recognize any painting in the image and the user is notified: UC_02.1 – PaintingNotRecognized.

Use case name	PaintingNotRecognized
Use case ID	UC_02.1
Participating actors	User
Preconditions	The user initiated the image processing task and is now waiting for a result.
Flow of events	1. The system cannot find any painting match in the captured image. 2. The system notifies the user by sending him a message.
Postconditions	The user is correctly notified and redirected to the home page.

Use case name	PlaceGuideModel
Use case ID	UC_03
Participating actors	User
Preconditions	The system has correctly recognized a painting in the user captured image.

Flow of events	1. The user points the camera in the scene and lets the system scan surfaces. 2. The user taps on a surface and attempts to place the 3D model of the virtual guide.
Postconditions	The guide model has successfully been placed and the user can now see it through their device's camera.
Exceptions	The surface on which the user tapped his finger is not a suitable area for the placement of the guide model and the user is notified: UC_03.1 – UnsuitableSurface.

Use case name	UnsuitableSurface
Use case ID	UC_03.1
Participating actors	User
Preconditions	The user has attempted positioning the guide model in the virtual environment.
Flow of events	1. The system does not place the guide model due to the tapped surface nod being suitable and notifies the user.
Postconditions	The user is correctly notified and has the ability to try again.

Use case name	StartNarration
Use case ID	UC_04
Participating actors	User
Preconditions	The user has successfully placed the guide model in the virtual environment.
Flow of events	1. The user starts the narration by tapping on the UI "Play" button. 2. The guide begins narrating the scene to the user.
Postconditions	The user is successfully listening to the augmented narration while the details of the painting are projected in the virtual environment.

Use case name	StopNarration
Use case ID	UC_05
Participating actors	User
Preconditions	The user has previously started the augmented narration of the painting.
Flow of events	1. By using the interface, the used taps on the "Stop" button.
Postconditions	The voice narration stops correctly, and the augmented details are still displayed in the scene.

Use case name	SkipAhead
Use case ID	UC_06
Participating actors	User
Preconditions	The user has previously started the augmented narration of the painting.
Flow of events	1. By using the interface, the used taps on the “Next” button.
Postconditions	The narration has been successfully interrupted and restored to the next segment; any displayed image has been removed.

Use case name	Backtrack
Use case ID	UC_07
Participating actors	User
Preconditions	The user has previously started the augmented narration of the painting.
Flow of events	1. By using the interface, the used taps on the “Previous” button.
Postconditions	The narration has been successfully interrupted and restored to the previous segment; any displayed image has been removed.

Use case name	AddPainting
Use case ID	UC_08
Participating actors	Administrator
Preconditions	The administrator has correctly accessed the system’s data storage console.
Flow of events	1. By using the system’s console, the administrator adds the information for a new painting for the system to recognize and narrate. 2. The administrator updates the information and confirms the procedure.
Postconditions	The new painting’s information has been successfully added to the system’s data storage and the corresponding painting will now be successfully recognized.

Use case name	UpdatePainting
Use case ID	UC_04
Participating actors	Administrator
Preconditions	The administrator has correctly accessed the system’s data storage console, and the painting to be updated exists in the system’s database.

Flow of events	1. By using the system's console, the administrator selects the painting to update. 2. The administrator updates the information and confirms the procedure.
Postconditions	The painting's information has successfully been updated and any will be used by any future narration.

Use case name	RemovePainting
Use case ID	UC_09
Participating actors	Administrator
Preconditions	The administrator has correctly accessed the system's data storage console, and the painting to be removed exists in the system's database.
Flow of events	1. By using the system's console, the administrator selects the painting to update and confirms the procedure.
Postconditions	The painting's information has successfully been removed and it will not be accessible by any narration in the future.

3.3 USER INTERFACE MOCK-UPS

According to both the identified requirements and the use case model, a first user interface prototype has been constructed: three main windows provide the system with the most essential functionalities; the first window acts as the home screen of the application: it contains a banner image and a button to start up the device's camera. The second window is used by the system to capture and then automatically process the image in the background, while the final window is where the storytelling takes place: the image represents the augmented painting, while the face acts as the talking virtual guide.

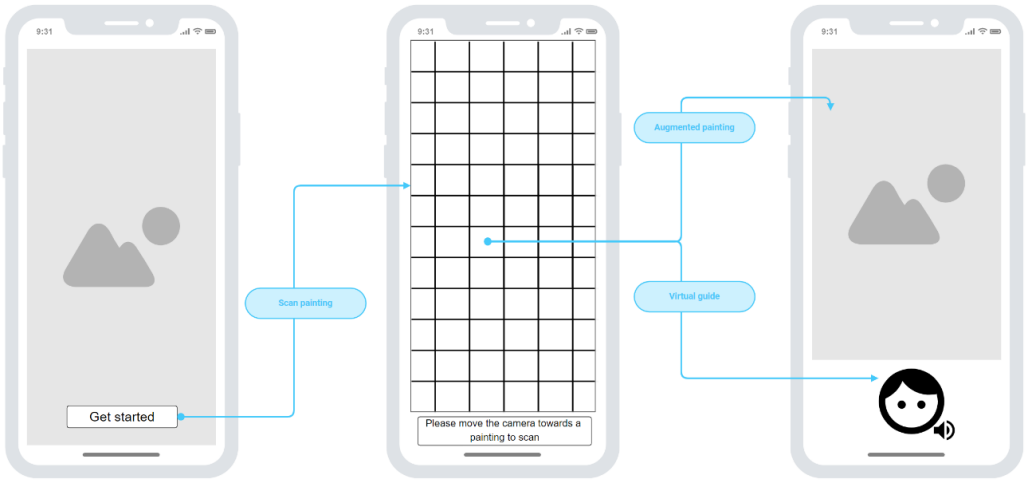


Figure 3.3 - First UI prototype

In this second UI prototype, the first two windows are the same, but we felt like the user should have more control over the image processing steps, so we deemed necessary the addition of a fourth window, where the captured image is previewed. This allows the user to cancel the operation or manually initiate the image processing step by uploading the image to a remote server.

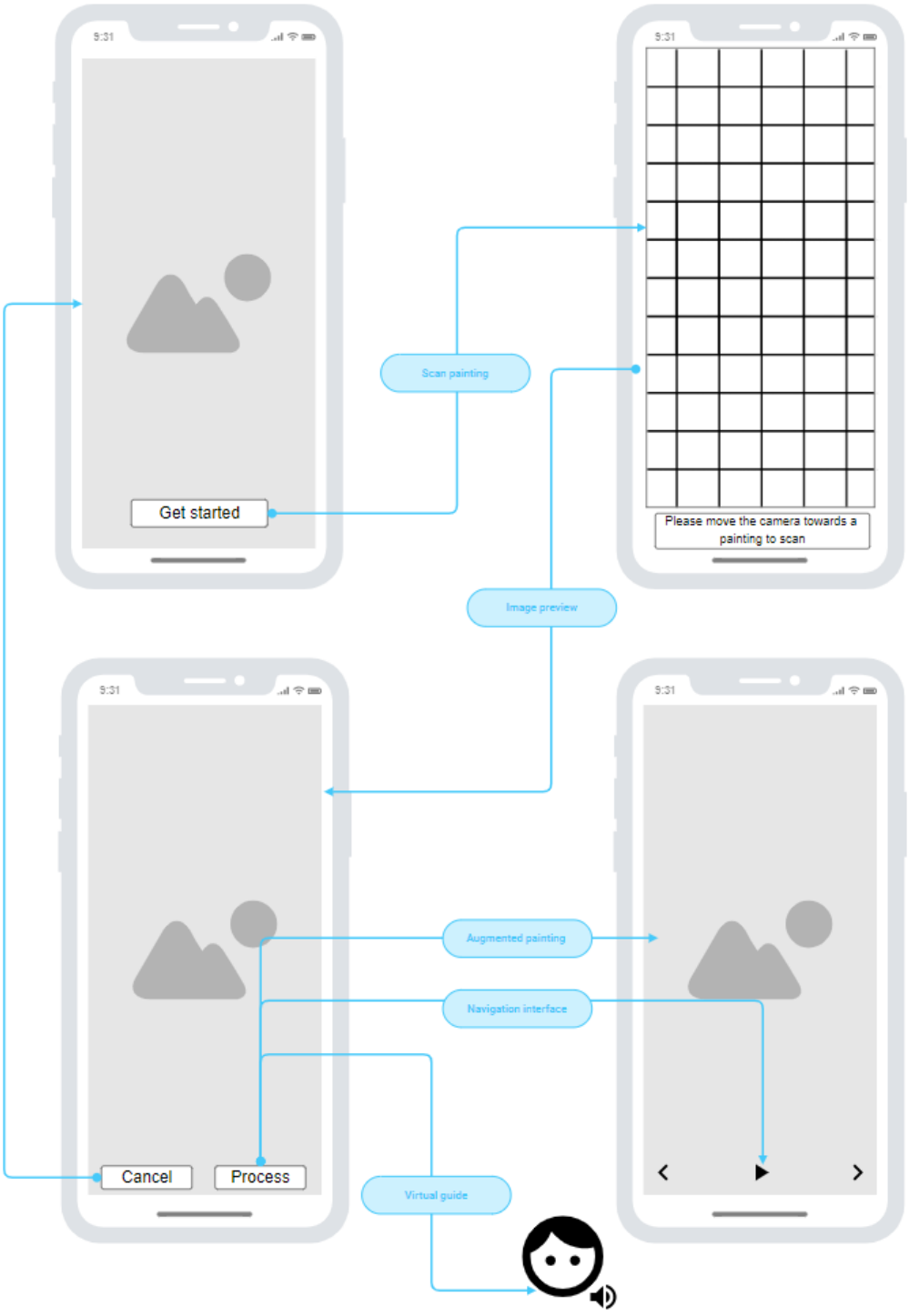


Figure 3.4 - Second and final UI prototype

4. APPLICATION DESIGN

4.1 ARCHITECTURE OVERVIEW

Following the initial requirements analysis and the system's goals, a set of design goals for the application has been identified; these will act as guidelines during the development, in order to ensure the implementation will not shift away from its original design views.

- **DG_01: Accessibility**: Ensuring the application is targeted towards a wide spectrum of users is a high priority;
- **DG_02: Reliability**: The information provided by the system must always be reliable;
- **DG_03: Usability**: The application's user interface must be simple and friendly to newcomers.

The general architecture of CiceroAR is based on a client-server model, in which many clients request and receive services from a centralized server (or servers). Client devices provide an interface to allow a user to request services of the server and to display the results the server returns, while servers wait for requests to arrive from clients and then respond to them.

At the hardware level, a distinction between three devices is made: the user's smartphones acts as the client side of the application, where the Image component encapsulates the camera control and the network functionalities, while the AR Narration component is where the actual narration of the painting takes place.

On one of the web hosts, the Node.js server acts as a middle-ground between the client and the image recognition server. The Node API component manages the network requests, while the Database component handles the persistent information associated to each painting.

On the second web host, a REST API, realized via the djangoRestFramework module, is used to interface the Django AI server with the Node server. Here the neural network is encapsulated in the ImageDetector component, in its own class which provides the methods necessary to run a pre-trained inference graph on the received image and output a title and a confidence score. If this confidence score, expressed as the probability of a specific painting being present in the scene, is greater than a pre-set threshold, a positive feedback is sent back to the Node server.

The two web servers could potentially be deployed onto the same machine or as a unique module; the reason they have been implemented as two distinct entities is just to keep a logic separation between the components.
The following UML deployment diagram depicts the three devices and their interaction via the HyperText Transfer Protocol, HTTP.

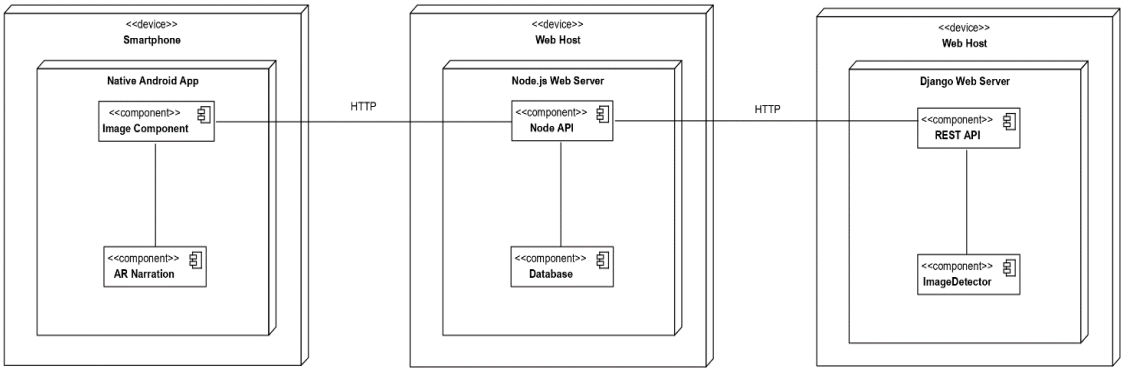


Figure 4.1 - UML deployment diagram of the system's architecture

This second diagram illustrates the processing of an image in CiceroAR as a list of steps, from the moment it is captured all the way to when the information relative to the recognized painting is retrieved and returned to the user.

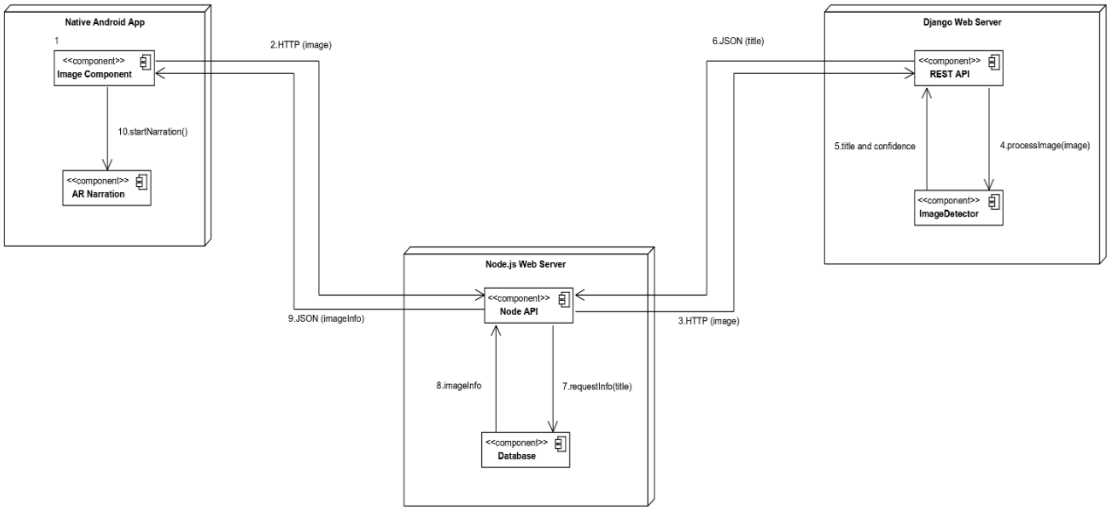


Figure 4.2 - UML deployment diagram of the image processing steps

1. User starts up the application and captures and image with the smartphone's integrated camera.
2. The captured image is sent to the Node.js server, using an HTTP multipart form-data request.

3. Once received, the image is forwarded from the Node API to Django server, via the exposed REST API.
4. The Django server receives the image and processes it via the underlying neural network, in the ImageDetector component.
5. A title and a confidence score are output for the received image by the neural network. It then passes them to the REST API component.
6. Response is then sent back to the Node.js server as a JSON object containing the two fields.
7. The Node.js server retrieves the information associated to the identified painting from a MongoDB database.
8. Any retrieved information is sent to the Node API component.
9. The information is sent back to the Android client.
10. The control is finally sent to the AR Component, which proceeds with the narration of the painting.

4.1.1 BOUNDARY USE CASES AND SOFTWARE EXCEPTIONS

After identifying the system's requirements, use cases and its general architecture, we still need to describe its boundary conditions and decide how the system is started, initialized and shut down and how it deals with different kinds of failure.

The first newly identified use cases depict the system's start up and shut down operations of the server components: they are performed by a system's administrator on the server machine. The client's start up and shut down operations are considered trivial and will not be further discussed.

Use case name	StartUpNodeServer
Use case ID	UC_B01
Participating actors	Administrator
Preconditions	The Node server is not currently running on the server machine.
Flow of events	1. The administrator executes the "node index.js" command on the system' s console.
Postconditions	The Node server has been initialized successfully and can now receive image requests from user devices.

Use case name	StartUpDjangoServer
Use case ID	UC_B02
Participating actors	Administrator
Preconditions	The Django server is not currently running on the server machine.
Flow of events	1. The administrator executes the “node index.js” command on the system’ s console.
Postconditions	The Django server has been initialized successfully and can now receive image requests from the Node server.

Use case name	ShutDownNodeServer
Use case ID	UC_B03
Participating actors	Administrator
Preconditions	The Node server has been previously started up and is running on the server machine.
Flow of events	1. The administrator shuts down the Node.js process via the system’s console.
Postconditions	The Node server has been successfully shut down and no further requests can be processed.

Use case name	ShutDownDjangoServer
Use case ID	UC_B04
Participating actors	Administrator
Preconditions	The Django server has been previously started up and is running on the server machine.
Flow of events	1. The administrator shuts down the Django process via the system’s console.
Postconditions	The Django server has been successfully shut down and no further requests can be processed.

The main faults CiceroAR could encounter during its typical flow of interaction are the following:

- A network error, due to which one or more connections between a user’s device and the Node.js server are interrupted;
- A network error, due to which one or more connections between the Node.js server and the Django server are interrupted, in the case these are deployed onto different remote machines;

- A server error, due to which one or more components on a web server are abruptly shut down: in this occasion the image processing steps may not successfully execute, or data corruption may take place when storing the information of a new painting.

In order to handle network errors between the client and the Node.js server, the Android device provides a pop-up error message informing the user of the fault; the captured image sent to the server gets lost and the user must repeat the operation from the beginning. Analogously, if a connection error between the Node server and the Django server takes place, the image is also discarded and not processed.
For the validation of the persistent storage's state after an abrupt shutdown, we introduce an exception use case, "CheckDataIntegrity", performed by a system administrator and used to check the database state.

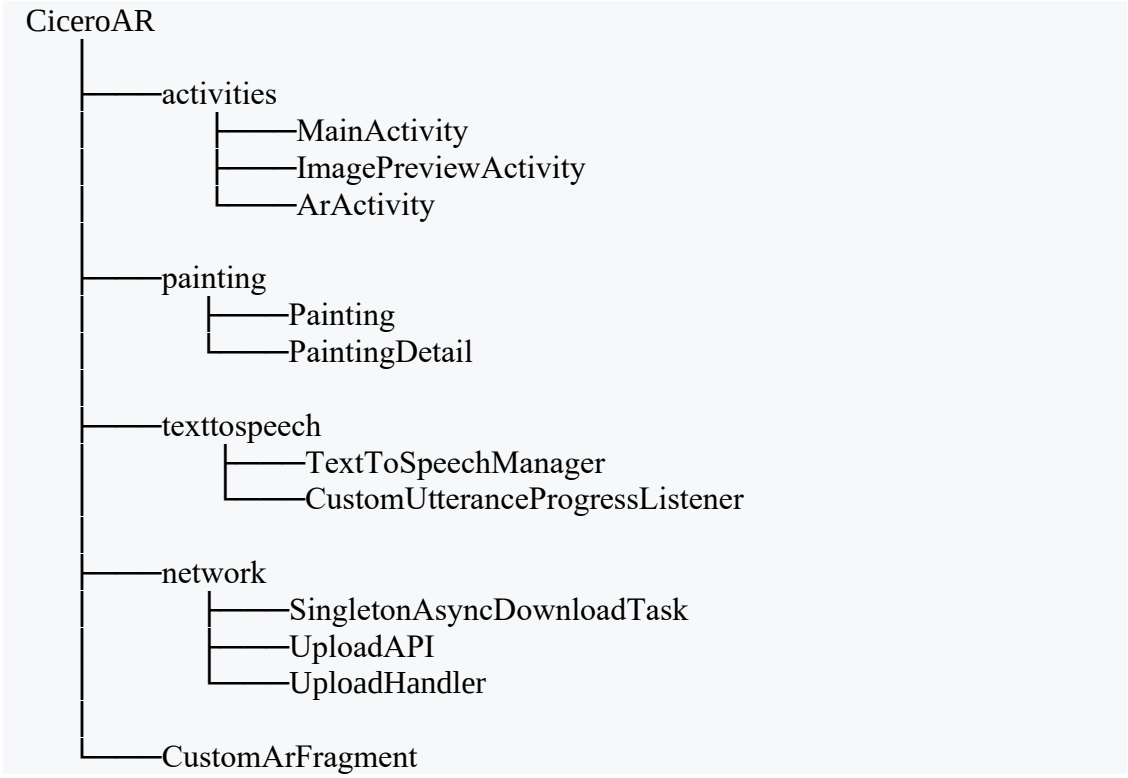
Use case name	CheckDataIntegrity
Use case ID	UC_E01
Participating actors	Administrator
Preconditions	The Node server has been previously started up and is running on the server machine.
Flow of events	1. The administrator executes the "checkDataIntegrity" command on the system' s console. 2. The system performs a series of test queries on the recently manipulated data in order to detect any corruption.
Postconditions	The administrator receives a brief report of the data's state.

4.2 ANDROID CLIENT

This section is dedicated to the client side of the system, a native Android application written in Java using the Android Studio environment, with SDK version 28 (Android 9, Pie) and Java version 8. It is executed on the user's device and handles the AR component of the system, as well as the TTS storytelling. The list of supported devices is specified in the ARCore documentation and the smartphone we used to test the system is a Samsung Galaxy S10^{+}.

4.2.1 PACKAGES AND CLASSES

CiceroAR's client is structured in a series of packages and classes, based on the core functionalities of the system; the following diagram summarizes this division.



The activities package includes the three Android activities composing the application; each of these is paired with an XML layout file, in which the UI and its styling attributes are specified. The MainActivity class provides the home screen functionalities and the access to the camera component, the ImagePreviewActivity class is used to display to the user the preview of the captured image and handle its upload and the ArActivity class handles the augmented environment and the thread communication.

The painting package acts as a model for the structure of the information associated to each painting and is used in order to correctly map the data received from the Node server to a plain old Java object (POJO).

The texttospeech package handles the life cycle of the objects managing the voice storytelling by specifying a progress listener and sending message to the main thread during key moments of the narration.

Finally, the network package contains the classes that handle the upload of the image file to the Node server and listen for a response.

The previous package diagram can be expanded by using a more complete UML class diagram, in which each class's variables, methods and associations are highlighted.

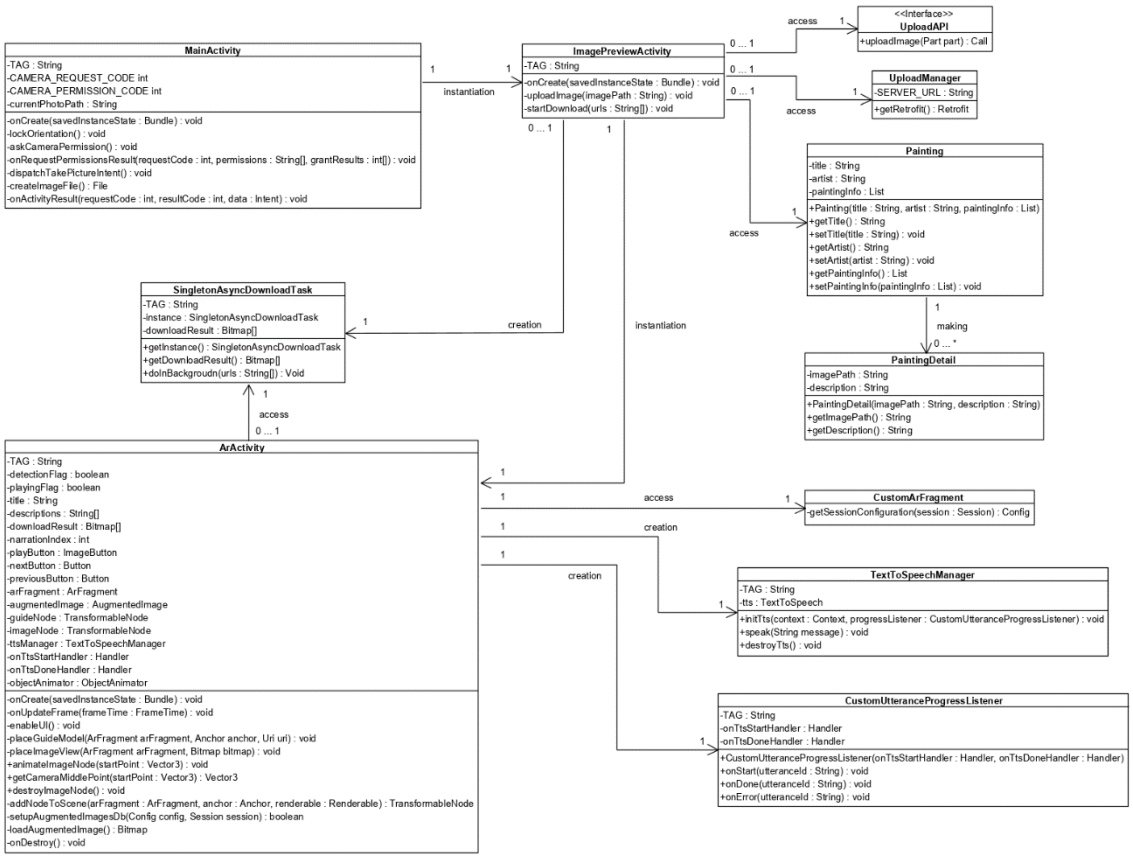


Figure 4.3 - UML class diagram of the client packages

4.2.2 ARCore and Sceneform

The Augmented Reality functionalities are provided by the ARCore SDK, Google's AR solution. By using various APIs, it allows smartphones to detect the surrounding environment, understand the world and interact with its information; ARCore is based on three key technologies to integrate the virtual content into the real world, as seen through the smartphone's camera:

- Six degrees of freedom allows the device to understand and track its position relative to the world;
- With environmental understanding, the device can detect the size and location of surfaces;
- Light estimation allows the device to estimate the environment's current lighting conditions.

ARCore calculates position and orientation of a device by detecting feature points in the captured images; the AR application looks for clusters of feature points to lie on common horizontal or vertical surfaces, and identifies them as planes, making them available to the application; it also defines each plane's information, which can be used to place virtual objects on flat surfaces. Another useful feature offered by this SDK is lighting estimation: based on the lighting of the camera image's environment, ARCore can light up objects accordingly and strengthen the overall augmented experience. Once a virtual object has been placed, the user can move the camera around it or away from it without affecting its tracking.
One of the main limitations of the library, and of markerless AR in general, is the difficulty of tracking a texture-less surface, such as a blank white wall; for this reason, the augmented images projected in the scene are anchored on top of the painting, which is tracked by the native ARCore AugmentedImage component, after being recognized.

When it comes to the handling of the 3D environment, the choice was between a fully-fledged game engine, like Unity or Unreal Engine, and an external Android library, such as Google's Sceneform. We decided for the latter, since this allowed us to keep a higher control over the whole application development; furthermore, this would make future maintenance or additions easier. Sceneform makes it straightforward to render 3D scenes into AR and non-AR applications, without having to learn OpenGL; it includes:

- A high-level scene graph API;

- A realistic physically based rendered provided by Filament;
- An Android Studio plugin for importing, viewing and building 3D assets.

The ArActivity class handles the virtual environment and the AR elements of the scene; the first step is to retrieve an ArFragment object from the activity's layout, in order to access its components; then a OnUpdateListener and a OnTapListener are associated to it. The on-tap listener is in charge of generating the 3D model of the virtual guide when the user taps on suitable surface on their device screen; once the model has been placed, any new tap of the finger will update its position. The following code snippet shows how the ArFragment object is retrieved and how its properties and listeners are set up.

```
arFragment = (CustomArFragment) getSupportFragmentManager().findFragmentById(R.id.sceneform_fragment);
assert arFragment != null;
arFragment.getPlaneDiscoveryController().hide();
arFragment.getArSceneView().getScene().addOnUpdateListener(this::onUpdateFrame);
arFragment.setOnTapArPlaneListener((HitResult hitResult, Plane plane, MotionEvent motionEvent) -> {
    Anchor anchor = hitResult.createAnchor();
    AnchorNode anchorNode = new AnchorNode(anchor);
    anchorNode.setParent(arFragment.getArSceneView().getScene());
    placeGuideModel(arFragment, anchor, Uri.parse("guide.sfb"));
});
```

Figure 4.4 - Code snippet . ArFragment and OnTapListener

The update listener, on the other hand, is a method containing the behaviour that will be executed every time the camera frame gets updated: initially, the system retrieves the collection of the objects that are currently being tracked in the environment; in this list we look for the painting that has been previously recognized by the AI component, in order to "recognize" it a second time and use it as an anchor point for the augmented elements. Once this second recognition takes place the navigation UI gets enabled and it shows up on the user's screen; then, by changing the value of a boolean flag, this behaviour of the function is ignored and, during each frame update, if any 2D augmented image has been placed in the scene, its rotation is updated to face the user's device at all times. This behaviour is highlighted in the next figure.

```java
/**
 * Method called on every frame update  on the scene. It initially liiks for the image in the scene
 * and when this is identified it proceeds to create the guide model and start the TTS narration
 * Finally it updates the rotation of the objects in the scene to face the camera in every instant
 *
 * @param frameTime the object containing the information for the current frame
 */
private void onUpdateFrame(FrameTime frameTime) {
    Frame frame = arFragment.getArSceneView().getArFrame();
    assert frame != null;

    if(detectionFlag) {
        Collection<AugmentedImage> augmentedImages = frame.getUpdatedTrackables(AugmentedImage.class);
        augmentedImages.forEach(augmentedImage -> {
            if(augmentedImage.getTrackingState() == TrackingState.TRACKING &&
                    augmentedImage.getName().equals(title)) {
                this.augmentedImage = augmentedImage;
                detectionFlag = false;
                enableUI();
            }
        });
    }

    // Rotates the image displayed to look at the viewer at all times
    if (imageNode != null) {
        Vector3 cameraPosition = arFragment.getArSceneView().getScene().getCamera().getWorldPosition();
        Vector3 imagePosition = imageNode.getWorldPosition();
        Vector3 imageDirection = Vector3.subtract(cameraPosition, imagePosition);
        Quaternion imageLookRotation = Quaternion.lookRotation(imageDirection, Vector3.up());
        imageNode.setWorldRotation(imageLookRotation);
    }
}
```

Figure 4.5 - Code snippet - onUpdateFrame

When it comes to the rendering of objects in an augmented environment, Sceneform makes a difference between two different kinds of "renderables": a ModelRenderable is used to manage the behaviour of 3D objects, while a ViewRenderable handles 2D images or sprites in the scene; we use the first class when generating the virtual guide model and the second to place and animate the painting's detail images. The way these 2D images are generated and destroyed is managed by two Handler objects that, as we will be discussing later in the 4.2.4 Text-To-Speech paragraph, will receive messages by the TTS thread during the storytelling and then proceed to create a pose in the centre of the image, in which to anchor the image or remove any previous reference to the image. The following two figures display three methods used in the ArActivity class, responsible of building the Model and View renderables respectively and then add the corresponding node in the scene.

```java
/**
 *Builds the model renderable object for the guide model and places it into the scene
 *
 * @param arFragment the fragment
 * @param anchor the anchor used to track the renderable
 * @param uri uri pointing to the 3D model .sfb file
 */
private void placeGuideModel(ArFragment arFragment, Anchor anchor, Uri uri) {
    if(guideNode != null) {
        arFragment.getArSceneView().getScene().removeChild(guideNode);
        guideNode.setParent(null);
        guideNode = null;
    }
    ModelRenderable.builder()
            .setSource(Objects.requireNonNull(arFragment.getContext()), uri)
            .build()
            .thenAccept(modelRenderable -> guideNode = addNodeToScene(arFragment, anchor, modelRenderable))
            .exceptionally(throwable -> {
                Toast.makeText(getApplicationContext(), "Error: " + throwable.getMessage(), Toast.LENGTH_SHORT).show();
                return null;
            });
}
```

Figure 4.6 - Code snippet - ModelRenderable creation

```java
/**
 * Builds the view renderable for the scene by using a reference to a standard android ImageView,
 * sets its image to the one received and places it into the scene
 *
 * @param arFragment the fragment
 * @param bitmap the bitmap containing the image data
 */
private void placeImageView(ArFragment arFragment, Bitmap bitmap) {
    destroyImageView();

    ViewRenderable.builder()
            .setView(arFragment.getContext(), R.layout.ar_layout)
            .build()
            .thenAccept(viewRenderable -> {
                imageRenderable = viewRenderable;
                imageRenderable.setShadowCaster(false);
                imageRenderable.setShadowReceiver(false);
                ImageView imageView = (ImageView) imageRenderable.getView();
                imageView.setImageBitmap(bitmap);

                // Creates a pose in the center of the augmented image, shifts it down slightly,
                // places an anchor in that point and uses it to add the image node to the scene
                Pose imagePose = augmentedImage.getCenterPose();
                imagePose = Pose.makeTranslation(0, -0.2f, 0).compose(imagePose);
                Anchor startAnchor = Objects.requireNonNull(arFragment
                        .getArSceneView()
                        .getSession())
                        .createAnchor(imagePose);
                imageNode = addNodeToScene(arFragment, startAnchor, imageRenderable);
                animateImageNode(imageNode.getWorldPosition());
            })
            .exceptionally(throwable -> {
                Toast.makeText(getApplicationContext(), "Error generating the image view: " +
                        throwable.getMessage(), Toast.LENGTH_SHORT).show();
                return null;
            });
}
```

Figure 4.7 - Code snippet - ViewRenderable creation

```java
/**
 * Creates an anchor node, sets his parent and renderable, and places it into the scene
 *
 * @param arFragment the fragment
 * @param anchor the anchor used to track the renderable
 * @param renderable the renderable to add in the scene
 * @return the created node's reference
 */
private Node addNodeToScene(ArFragment arFragment, Anchor anchor, Renderable renderable) {
    AnchorNode anchorNode = new AnchorNode(anchor);
    anchorNode.setParent(arFragment.getArSceneView().getScene());
    Node node = new Node();
    node.setRenderable(renderable);
    node.setParent(anchorNode);
    arFragment.getArSceneView().getScene().addChild(anchorNode);
    return node;
}
```

Figure 4.8 - Code snippet - Node creation

The placement of the ModelRenderable is relatively straight forward: first the 3D model of the guide is retrieved by using a URI and it is assigned to the renderable which is then placed in the scene by using the addNodeToScene() method. A few more steps are required when dealing with ViewRenderable objects: to begin, are based on Android's standard ImageView component, used to display an image in a normal UI layout, so a layout file detailing the specifications of the component is required; once this has been retrieved, we associate the actual image to it, passed as a Bitmap object to the method, and remove casted and received shadows to it, in order to make the view more clean and pleasant to look at. The position in which to anchor the renderable corresponds to the centre of the augmented image previously recognized: said measurements can be obtained via the getCenterPose() method called on the AugmentedImage object. A small shift downwards on the y axis is required in order to properly centre the image, since otherwise its lower central point is in the centre of the augmented image. Finally, the image can be actually placed, by creating an anchor in the newfound position and calling the addNodeToScene() method.

At this point, once it has been placed, the image sprite begins translating forward, due to an animation realized by using an ObjectAnimator object: firstly the destination point is calculated, based on a middle point between the image current position and the camera pose, then a LinearInterpolator is used to gradually move the sprite forward; any pause of the narration would pause the animation as well. Once both the animation and the

narration segment are complete, the second handler destroys the image node, which is no longer needed and the storytelling proceeds to the next segment (or stops).

```java
/**
 * Calculates the middle point between the image node and the camera
 * and returns it
 *
 * @param startPoint vector3 containing the image's initial position
 * @return the endpoint for the animation transition
 */
private Vector3 getCameraMiddlePoint(Vector3 startPoint) {
    // Creates a pose in the current camera position and places and anchor in the middle point
    // between the image node and the camera
    Pose cameraPose = Objects.requireNonNull(arFragment.getArSceneView().getArFrame()).getCamera().getPose();
    float distance = (float) Math.sqrt(
            Math.pow((startPoint.x - cameraPose.tx()), 2) +
            Math.pow((startPoint.y - cameraPose.ty()), 2) +
            Math.pow((startPoint.z - cameraPose.tz()), 2)
    ) / 2;
    cameraPose = Pose.makeTranslation(0f, 0f, -distance).compose(cameraPose);
    Anchor endAnchor = Objects.requireNonNull(arFragment
            .getArSceneView()
            .getSession())
            .createAnchor(cameraPose);
    AnchorNode endNode = new AnchorNode(endAnchor);
    endNode.setParent(arFragment.getArSceneView().getScene());
    return endNode.getWorldPosition();
}
```

Figure 4.9 - Code snippet - Distance calculation

```java
/**
 * Animates a previously generated image node in the scene by moving it forward
 * towards the user camera.
 *
 * @param startPoint vector3 containing the image's initial position
 */
private void animateImageNode(Vector3 startPoint) {
    Vector3 endPoint = getCameraMiddlePoint(startPoint);

    // Animates the image forward in the environment
    objectAnimator = new ObjectAnimator();
    objectAnimator.setAutoCancel(true);
    objectAnimator.setTarget(imageNode);
    objectAnimator.setObjectValues(startPoint, endPoint);
    objectAnimator.setPropertyName("worldPosition");
    objectAnimator.setEvaluator(new Vector3Evaluator());
    objectAnimator.setInterpolator(new LinearInterpolator());
    objectAnimator.setDuration(20000);
    objectAnimator.start();
}
```

Figure 4.10 - Code snippet - Image animation

In order to provide a better user experience, we decided to implement user interface functionalities to navigate through the narration; these include a pause/resume button and two buttons to move to the previous and next narration segments. The user interface managed by this class is therefore made up of two Button objects and one ImageButton object: the two buttons allow the user to skip to the next or previous narration segments, while the image button handles the pause/resume functions and its icon changes accordingly. Initially the UI is disabled, up until the system recognizes the painting as a suitable anchoring surface.

```
/**
 * Enables the UI buttons after the image has been recognized
 */
private void enableUI() {
    playButton.setVisibility(View.VISIBLE);
    playButton.setBackground(null);
    nextButton.setVisibility(View.VISIBLE);
    previousButton.setVisibility(View.VISIBLE);
}
```

Figure 4.11 - Code snippet - UI enabler

```
nextButton = findViewById(R.id.next_button);
nextButton.setVisibility(View.GONE);
nextButton.setOnClickListener(v -> {
    narrationIndex = (narrationIndex >= descriptions.length - 1 ? descriptions.length - 1 : narrationIndex + 1);
    destroyImageView();
    if(playingFlag) {
        ttsManager.stop();
        ttsManager.speak(descriptions[narrationIndex]);
    }
});

previousButton = findViewById(R.id.previous_button);
previousButton.setVisibility(View.GONE);
previousButton.setOnClickListener(v -> {
    narrationIndex = (narrationIndex > 0 ? narrationIndex - 1 : 0);
    destroyImageView();
    if(playingFlag) {
        ttsManager.stop();
        ttsManager.speak(descriptions[narrationIndex]);
    }
});
```

Figure 4.12 - Code snippet - Logic behind the next/previous buttons

```
// UI buttons listeners setup
playButton = findViewById(R.id.play_pause_button);
playButton.setVisibility(View.GONE);
playButton.setOnClickListener(v -> {
    if (!playingFlag) {
        /*
         * TTS is currently paused
         * Changes the ImageButton icon to the pause icon and resumes the narration
         */
        Drawable icon = getDrawable(android.R.drawable.ic_media_pause);
        playButton.setImageDrawable(icon);
        playingFlag = true;

        ttsManager.speak(descriptions[narrationIndex]);
    } else {
        /*
         * TTS is currently playing
         * Changes the ImageButton icon to the play icon and stops the narration
         */
        Drawable icon = getDrawable(android.R.drawable.ic_media_play);
        playButton.setImageDrawable(icon);
        playingFlag = false;
        ttsManager.stop();
        if(objectAnimator != null) {
            objectAnimator.pause();
        }
    }
});
```

Figure 4.13 - Code snippet - Logic behind the pause/resume button

The logic behind these buttons is very simple: starting with the pause and resume ImageButton, a boolean flag is used to keep track of the playing state of the TTS object: if this flag is set to false, the narration is currently pause and the button icon is the “play” icon, so by tapping it, its icon switches to the “pause” symbol and the narration begins by repeating the last segment where it was interrupted (the details of this decision are discussed ahead). If the playing flag is set to true, the button’s icon switches to the “play” symbol and the TTS narration halts.

The next and previous button share a similar behaviour, with the exception of the icon change; additionally, they perform various checks on the value of the current narration index, to avoid ending up out of bounds.

4.2.3 *NETWORKING AND PAINTING INFORMATION*

The most crucial networking activity in CiceroAR is the handling of the response received from the Node.js server, following the image upload. The details about the response's structure will be provided in the Node server section, in the 4.3 paragraph.
The following code snippet, contained in the ImagePreviewActivity class shows this behaviour:

```
@Override
public void onResponse(@NonNull Call<Painting> call, @NonNull Response<Painting> response) {
    progressBar.setVisibility(View.GONE);
    if(response.body() == null) {
        Log.e(TAG, "Response error: no body in response");
    }
    else {
        Toast.makeText(getApplicationContext(), "Image successfully uploaded", Toast.LENGTH_SHORT).show();
        List<PaintingDetail> paintingDetails = response.body().getPaintingDetails();

        List<String> urls = new ArrayList<>();
        List<String> descriptions = new ArrayList<>();
        paintingDetails.forEach(entry -> {
            urls.add(entry.getImagePath());
            descriptions.add(entry.getDescription());
        });
        new SingletonAsyncDownloadTask().getInstance().execute(urls.toArray(new String[0]));

        Intent intent = new Intent(getApplicationContext(), ArActivity.class);
        intent.putExtra("artist", response.body().getArtist());
        intent.putExtra("title", response.body().getTitle());
        intent.putExtra("descriptions", descriptions.toArray(new String[0]));
        startActivity(intent);
    }
}

@Override
public void onFailure(@NonNull Call<Painting> call, @NonNull Throwable t) {
    progressBar.setVisibility(View.GONE);
    t.printStackTrace();
    Toast.makeText(getApplicationContext(), "Error! No response from server", Toast.LENGTH_SHORT).show();
}
```

Figure 4.14 - Code snippet - Node server response handling

Since the response is in JSON format, we needed a way to map its data into a POJO: the Painting class fits this purpose and has been built to contain the names of the author and of the painting itself, as well as a list of PaintingDetail objects, in which we store the URL - description pairs received from the remote server. These pairs are made up of a unique locator for the image of a specific painting detail and its associated description to use during the storytelling process.
After receiving from the Node.js server a JSON response containing the painting information and mapping it to a Painting object, along with the resource URLs, the

system must download the images corresponding to each description fragment. In order to not block the main UI thread, this operation is performed asynchronously in the background, by a singleton class, SingletonAsyncDownloadTask, subclass of AsyncTask. Singleton is a creational design pattern, which ensures only one object of a class can exist at any time and provides a unique access point to it for any other object; in the context of our application, the ImagePreviewActivity class creates the unique instance after receiving the response from the server and starts up the download process. Once the narration begins, the ArActivity class accesses the previously created instance and retrieves the downloaded data, before using it to display the images in the scene.

```
@Override
protected Void doInBackground(String... strings) {
    if(instance == null) {
        return null;
    }
    try {
        downloadResult = new Bitmap[strings.length];
        for(int i = 0; i < strings.length; i++) {
            if(strings[i] == null) {
                downloadResult[i] = null;
               continue;
            }
            URL url = new URL(strings[i]);
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            connection.connect();
            InputStream inputStream = connection.getInputStream();
            BufferedInputStream bufferedInputStream = new BufferedInputStream(inputStream);
            downloadResult[i] = BitmapFactory.decodeStream(bufferedInputStream);
        }
    } catch (IOException e) {
        Log.e(TAG, "Error while retrieving the image: " + e);
    }
    return null;
}
```

Figure 4.15 - Code snippet - doInBackground method in the Singleton class

The doInBackground() method is the one responsible of the actual download: it is executed on a separate thread and for each URL resource contained in the server response creates a connection to the respective image and starts the download, saving the data in a Bitmap array.

4.2.4 *TEXT-TO-SPEECH*

In order to handle the Text-To-Speech narration, the integrated android.speech.tts module has been used; this provides the most common functionalities one would expect from a TTS library, such as the possibility to switch the language and the voice's pace, start and stop an utterance and saving a voice recording to a file. We used a wrapper class, TextToSpeechManager, to contain and handle the TTS object.

```
public class TextToSpeechManager {
    private static final String TAG = "TextToSpeechManager";
    private TextToSpeech tts;

    public void initTts(Context context, CustomUtteranceProgressListener progressListener) {
        tts = new TextToSpeech(context, status -> {
            if(status == TextToSpeech.SUCCESS) {
                int result = tts.setLanguage(Locale.US);
                if(result == TextToSpeech.LANG_MISSING_DATA || result == TextToSpeech.LANG_NOT_SUPPORTED) {
                    Log.d(TAG, "TTS initialization failed: the language is not supported or the data is missing");
                } else {
                    tts.setOnUtteranceProgressListener(progressListener);
                }
            } else {
                Log.e(TAG, "Error initializing the Text-To-Speech engine");
            }
        });
    }


    public void speak(String message) {
        String utteranceID = "Utterance:" + System.currentTimeMillis();
        tts.speak(message, TextToSpeech.QUEUE_ADD, null, utteranceID);
    }

    public void stop() {
        if(tts != null) {
            tts.stop();
        }
    }

    public void destroyTts() {
        if(tts != null) {
            tts.stop();
            tts.shutdown();
        }
    }
}
```

Figure 4.16 - Code snipper - TextToSpeechManager class

Each TTS utterance is executed asynchronously on a different thread, therefore the narration works by setting up a custom UtteranceProgressListener for the TTS object and using two handlers to receive messages from the its thread during key moments of the utterance life cycle.

```java
public class CustomUtteranceProgressListener extends UtteranceProgressListener {
    private static final String TAG = "CustomUtteranceProgressListener";
    private Handler onTtsStartHandler;
    private Handler onTtsDoneHandler;

    public CustomUtteranceProgressListener(Handler onTtsStartHandler, Handler onTtsDoneHandler) {
        this.onTtsStartHandler = onTtsStartHandler;
        this.onTtsDoneHandler = onTtsDoneHandler;
    }

    @Override
    public void onStart(String utteranceId) {
        Message ttsStartMessage = onTtsStartHandler.obtainMessage();
        Bundle bundle = new Bundle();
        bundle.putString("New utterance started. UtteranceID: ", utteranceId);
        ttsStartMessage.setData(bundle);
        onTtsStartHandler.sendMessage(ttsStartMessage);
    }

    @Override
    public void onDone(String utteranceId) {
        Message ttsStartMessage = onTtsDoneHandler.obtainMessage();
        Bundle bundle = new Bundle();
        bundle.putString("New utterance successfully completed. UtteranceID", utteranceId);
        ttsStartMessage.setData(bundle);
        onTtsDoneHandler.sendMessage(ttsStartMessage);
    }

    @Override
    public void onError(String utteranceId) {
        Log.d(TAG, "Error during the handling of the Text-To-Speech utterance; utteranceID: " + utteranceId);
    }
}
```

Figure 4.17 - Code snippet - CustomUtteranceProgeressLsitener

This class overrides three inherited methods: onStart() is called as soon as the speak() method is called on the TTS object and sends a message to the corresponding handler waiting on the main thread; analogously, the onDone() method is called when an utterance is successfully completed and sends a message to another handler. Finally, onError() is called when something goes wrong during the processing of an utterance; the method simply logs the error.

The reason why this messaging system is necessary in the first place, is because all operations influencing the augmented elements of the scene must be performed on the main thread on which the application is running, or UI thread.

The first handler, onTtsStartHandler, receives a message as soon as the speak() method on the TextToSpeech object has been called: it proceeds to generate the image corresponding to the segment which is being narrated and animates it forward in the environment, as previously stated. The second handler, onTtsDoneHandler, receives a message when an utterance has been successfully completed, without any errors, and it destroys the previously generated image node in the scene.

```java
/**
 * Handler responsible to manage the message sent from the TTS thread when an utterance is started.
 * It places the image in the scene and interpolates it between
 * the position in which it has been generated and a forward point
 */
private final Handler onTtsStartHandler = new Handler(Looper.getMainLooper()) {
    @Override
    public void handleMessage(@NonNull Message message) {
        Log.d(TAG, "onTtsStartHandler. Message: " + message);
        if (downloadResult[narrationIndex] != null) {
            placeImageView(arFragment, downloadResult[narrationIndex]);
        }
    }
};


/**
 * Handler responsible to manage the message sent from the TTS thread when an utterance is completed
 * It performs a check on the next narration segments and if it is not the last one it initiates it
 */
private final Handler onTtsDoneHandler = new Handler(Looper.getMainLooper()) {
    @Override
    public void handleMessage(@NonNull Message message) {
        Log.d(TAG, "onTtsDoneHandler. Message: " + message);

        narrationIndex++;
        if(narrationIndex >=descriptions.length) {
            narrationIndex = 0;
            playingFlag = false;
            Drawable icon = getDrawable(android.R.drawable.ic_media_play);
            playButton.setImageDrawable(icon);
        } else {
            ttsManager.speak(descriptions[narrationIndex]);
        }
    }
};
```

Figure 4.18 - Code snippet - TTS Handlers

Since the native TTS module doesn't provide any pause functionalities we found ourselves with two main alternatives:

1. Make the TTS object record the narration to an audio file, using the native synthesizeToFile() method, and then employ a MediaPlayer object to control the narration at a static level;
2. On resume, make the narration start from the beginning of the current segment.

By using the first approach any future implementation of any kind of interaction mechanism would require a reengineering of the speech context, so we decided to proceed with the second approach, by carefully dividing the segments into smaller, independent parts.

4.3 NODE.JS SERVER

This section is dedicated to the Node.js web server, the external modules we used to integrate it with other components of our application and the employed data storage technologies.

4.3.1 NODE.JS

Node.js is a JavaScript runtime environment, entirely open-source and cross-platform; it executes JavaScript code outside of a web browser and is generally employed as a server-side scripting tool and therefore can enforce a "JavaScript everywhere" paradigm, unifying web-application development around a single programming language. The architecture is event-driven enabling development of fast and scalable web servers without the need of multithreading, by using callbacks to signal the completion of a task. Using events, allows a Node application to request to the operating system a notification upon the receiving of a particular event: while waiting the application is in a sleep state, allowing for a smarter handling of high traffic network scenarios.

While the base modules in Node provide a wide coverage of many server-side issues, there are countless external components that have been developed to work in synergy with Node during the years. Here is a list of the ones that we have used in combination with the native Node components:

- **cors:** Cross-Origin Resource Sharing authorization;
- **express:** GET/POST request behaviour;
- **multer:** storage of the image file;
- **mongodb:** database access;
- **fs:** file system access;
- **request :** request forwarding to the Django server.

In the following code snippet, we highlight one of the functions in the index.js file on the Node server: its purpose is to handle the post requests coming from the Android client of the system.

```
app.post('/upload', upload.single('image'), (req, res) => {
    var fs = require('fs');
    var request = require('request');
    var filename = './uploads/' + req.file.filename;


    /*
    * API request setup
    */
    const options = {
        method: 'POST',
        url: 'http://localhost:8000/upload/',
        headers: {
            'Content-Type': 'multipart/form-data'
        },
        formData: {
            'image': fs.createReadStream(filename)
        }
    };


    /*
    * Forwards the image to the REST API, sends the json response
    * body back to the Android client and deletes the image
    */
    console.log('Sending request to REST API...');
    request(options, (err, APIresponse, body) => {
        if(err) throw err;
        var json = JSON.parse(body);
        var title = json.title;

        if(title != null) {
            const MongoClient  = require('mongodb').MongoClient;
            const dbUrl = 'mongodb://localhost:27017/';
            MongoClient.connect(dbUrl, { useUnifiedTopology: true }, (err, db) => {
                var dbo = db.db('PaintingsInfo')
                dbo.collection('Paintings').findOne({ title: title}, (err, result) => {
                    if(err) throw err;
                    db.close;
                    res.json(result);
                });
            });
        } else {
            res.send(null);
        }

        fs.unlink(filename, (err) => {
            if(err) throw err;
        });
    });
});
```

Figure 4.19 - Code snippet - Node server post request handling

The options variable contains the information necessary in order to forward the request containing the image to the REST API; then the request is sent and the json body is retrieved: this contains the title of the painting, assuming the recognition has been successful. At this point an object used to access the underlying MongoDB database is created and the title is used as the key to retrieve the associated document which is then sent to the error. Once the process is completed, any reference to the received image is deleted, since we decided to store the image on the server before forwarding it, for testing and debugging purposes.

4.3.2 NoSQL and MongoDB

When facing the problem of storing the information associated to a painting, we encountered many alternatives, the main ones being a relational database, raw files or other, less common, storage solutions. Security has not really been a concern during the development, since the application does not provide any logging/sign up mechanisms and does not store any user sensible information; also, the painting data was retrieved from public domain, so no cryptography or hashing procedures were necessary. We decided to use a NoSQL database, MongoDB, to fill our need for storage; in NoSQL databases, data is not stored in the traditional tabular relations used in relational databases; such systems may still support SQL-like query languages or sit alongside SQL databases and for this reason the technology is often referred to as "Not only SQL". The main motivation behind this approach, which is seeing an increased usage in big data and real-time web applications, includes simplicity of the design, simpler "horizontal" scaling to clusters of machines, finer control over availability and limiting the object-relational impedance mismatch; the data structures used in NoSQL databases are different from those used by default in relational databases, making some operations faster in NoSQL. In distributed contexts, a NoSQL database offers a concept of "eventual consistency", in which database changes are propagated to all nodes "eventually", so queries for data might not return updated data immediately; this compromise in consistency is made in favour of availability, partition tolerance and speed.

MongoDB is one of the most widely used NoSQL databases; it is a document-oriented solution for data storage which is based on JSON-like documents with optional schemas: these documents easily map to objects in the application code, making the data easier to work with, especially when used in combination with Node.js on a server-side

application. Some of its known features include indexing of document fields, high availability of data with replica sets and load balancing.

```
const { MongoClient } = require('mongodb');
const dbUrl = 'mongodb://localhost:27017/';


MongoClient.connect(dbUrl, { useUnifiedTopology: true }, (err, db) => {
    if(err) throw err;
    console.log('Database created');

    var dbo = db.db('PaintingsInfo');
    dbo.createCollection("Paintings", (err, res) => {
        if(err) throw err;
        console.log('Collection created');
        db.close();
    });

    const fs = require('fs');
    var rawData = fs.readFileSync('data.json');
    var jsonData = JSON.parse(rawData);
    console.log(jsonData);

    dbo.collection('Paintings').insertMany(jsonData, (err, res) => {
        if(err) throw err;
        console.log('Number of entries created: ' + res.insertedCount);
        db.close();
    });
});
```

Figure 4.20 - Code snippet - Node server initial data setup

All of the initial data entries making up the database are contained in a JSON file as an array of objects; upon the first start-up of the Node server, this file is loaded into memory by using the external fs module and its entries are stored in the database, under the "Paintings" collection.

Example of a JSON data-entry/response in CiceroAR:

```
{
    “artist” : “Sandro Botticelli”,
    “title” : “Venere”,
    “paintingDetails” : [
        {
            “imagePath” :  null,
            “description”: “The main focus of the composition…”
        },
        {
            “imagePath” : “http://localhost:8080/paintings/Venere/Venus.jpg”,
            “description”: “The goddess is standing on a giant scallop shell…”
        },
        {
            “imagePath” :  “http://localhost:8080/paintings/Venere/Shell.jpg”,
            “description”: ”You may wonder why Venus is standing on a shell…”
        },
        {
            “imagePath” :  “http://localhost:8080/paintings/Venere/Zephyrus.jpg”,
            “description”:  “In the top left of the piece we can notice Zephyrus…”
        },
        {
            “imagePath” : “http://localhost:8080/paintings/Venere/Zephyrus.jpg”,
            “description”: ” The Hora herself might be a complementary version of…”
        }
    ]
}
```

Such an object is what is what gets sent back to the client in case of a successful recognition: the artist and title fields are self-explicatory, while the paintingDetails array contains the series of pairs that will make up the narration; all the images referenced in the imagePath fields are made available on the server machine as a static resource and can be accessed publicly.

An administrator of the system can add new paintings, either by creating a JSON array of painting or as a standalone JSON file and add them to the actual database.

4.4 DJANGO SERVER

In this last section dedicated to the main components of the application, we provide an in-depth overview of the Django AI server.

4.4.1 DJANGO AND TENSORFLOW

To provide the network functionalities necessary to interface the Node.js server to the Image Recognition component we used Django and the djangoRestFramework module, while, to create and train the neural network employed for the painting recognition, we made use of Google's TensorFlow library.

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design and has recently become a very popular way to quickly develop websites or write REST APIs, due to its security and speed. The architecture can be seen as a Model-View-Controller variation: it consists of an object-relational mapper (ORM) that mediates between data models, defined as Python classes, and a relational database (Model), a system for processing HTTP requests with a web templating system (View), and a regular-expression-based URL dispatcher (Controller).

We opted for Django since we needed a fast and simple way to interface ourselves with the AI component of the application; the djangoRestFramework module allowed us to quickly write an endpoint to do so, in order to give more thought and focus on the image recognition steps.

TensorFlow is an open source platform for Machine Learning developed by Google's Brain team. It can run on multiple CPUs and GPUs and provides a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in Machine Learning. The name derives from the operations that neural networks perform on multidimensional arrays, which are referred to as tensors.

The exhaustive documentation and various user resources have been trivial for our understanding of the problem since we had no previous experience with Machine Learning issues and frameworks. A pre-trained model was used to approach the recognition problem via transfer learning, a Machine Learning research problem that focuses on storing knowledge gained while solving one problem and applying it to a different one. The used model was previously trained on the COCO dataset, a large-scale object detection dataset, and is able to recognize hundreds of classes of common objects. By then feeding our small dataset to the network, the model was able to transfer

what it had learned from other object classes to the recognition of specific paintings in a scene.

The image recognition functionalities are wrapped into the ImageDetector.py; it contains the path to the frozen inference graph and the file containing the labels, as well as the ImageDetector class itself. The class includes two main methods, process_image(), and run_inference_for_single_image(); the first one retrieves and stores the inference graph and the received image file and then calls the run_inference_for_single_image() method.

```
"""
Runs the processing function and returns the results

@:returns the tuple created by _run_inference_for_single_image()
"""
def process_image(self):
    detection_graph = tf.Graph()
    with detection_graph.as_default():
        od_graph_def = tf.compat.v1.GraphDef()
        with tf.io.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid:
            serialized_graph = fid.read()
            od_graph_def.ParseFromString(serialized_graph)
            tf.import_graph_def(od_graph_def, name='')

    image = Image.open(os.path.join(MEDIA_ROOT, self.__image))
    image_np = self._load_image_into_numpy_array(image)
    return self._run_inference_for_single_image(image_np, detection_graph)
```

Figure 4.21 - Code snippet - process_image method

```
"""
Wrapper function to call the model and cleanup the outputs

@:param image the target image
@:param graph the detection graph used to process the image
@:returns a tuple containing the highest detection score
    and the corresponding class name detected for the image
"""
@staticmethod
def _run_inference_for_single_image(image, graph):
    category_index = label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS, use_display_name=True)

    with graph.as_default():
        config = tf.compat.v1.ConfigProto()
        config.gpu_options.allow_growth = True

        with tf.compat.v1.Session(config=config) as session:
            ops = tf.compat.v1.get_default_graph().get_operations()
            all_tensor_names = {output.name for op in ops for output in op.outputs}
            tensor_dict = {}
            for key in ['detection_scores', 'detection_classes']:
                tensor_name = key + ':0'
                if tensor_name in all_tensor_names:
                    tensor_dict[key] = tf.compat.v1.get_default_graph().get_tensor_by_name(tensor_name)
            image_tensor = tf.compat.v1.get_default_graph().get_tensor_by_name('image_tensor:0')

            output_dict = session.run(tensor_dict, feed_dict={image_tensor: np.expand_dims(image, 0)})
            output_dict['detection_classes'] = output_dict['detection_classes'][0].astype(np.uint8)
            output_dict['detection_scores'] = output_dict['detection_scores'][0]

    return output_dict['detection_scores'][0], [category_index.get(output_dict['detection_classes'][0]).get('name')]
```

Figure 4.22 - Code snippet – run_inference_for_single_image method

This method then uses the machine's GPU to run the inference graph and check for the presence of any previously set labels; it retrieves a list of detection classes with the relative detection scores, all sorted from biggest to smallest, and the first element is returned and used for a threshold check. The REST endpoint waits for a post request containing the image file, then instantiate an ImageDetector object and calls the process_image() method; if the recognition has been successful and the highest detection score is bigger than the pre-set threshold, a small JSON response containing the title of the painting is sent back to the Node server.

4.5 *FINAL SYSTEM AND CASE STUDY*

This last paragraph is dedicated to the final application's look and feel, and to the case study we decided to focus on in order to test the system and drive its development.

4.5.1 SYSTEM'S USER INTERFACE

The following figures extracted the running application depict the main pages composing the user interface, as well as some visual augmentations in the scene. The whole interface is pretty minimalistic: we felt like this was ideal for the type of application, since this way the functionalities are easily reachable, and the overall navigation is not ambiguous.

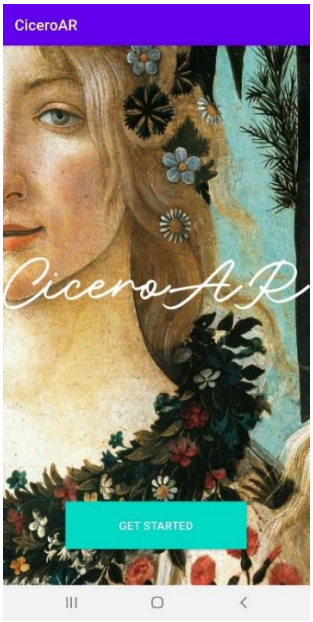


Figure 4.23 - Home screen

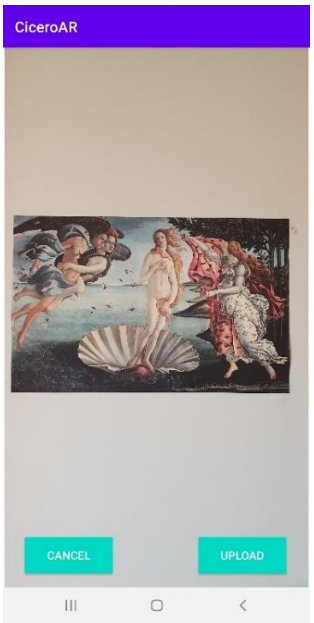


Figure 4.24 - Image preview window

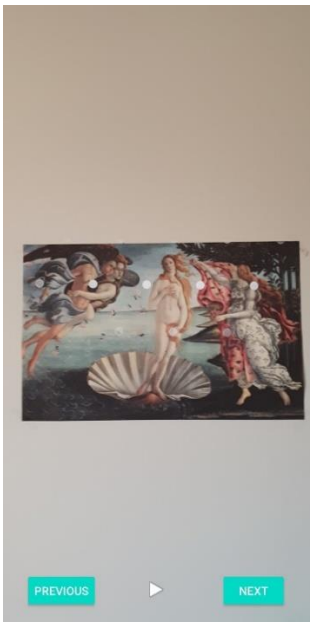


Figure 4.25 - UI before starting the storytelling

The first image depicts the home screen of the application: the “Get started” button is responsible to launch the camera and allow to user to capture an image. Once the image has been snapped, it is displayed as a preview; here the user can cancel the operation or process the image by uploading it. Finally, in the last figure, the system is ready to begin the narration process, after the user clicks on the start button. The white dots are used by the ARCore SDK to highlight detected surfaces and act as a visual confirmation for the scanning process; they could be disabled, however we noticed that, although this would make the UI more clean and pleasant, in this case the recognition would take longer, on average, since the user would have no idea about which part of the surface has been already detected.

These last images depict two visual augmentations, as they are being projected forward during the storytelling: as previously stated, these will rotate according to the user’s camera movement, in order to always face the spectator.

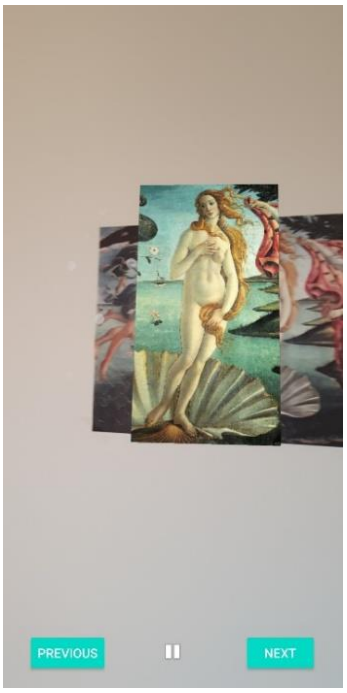


Figure 4.26 - Visual augmentation 1

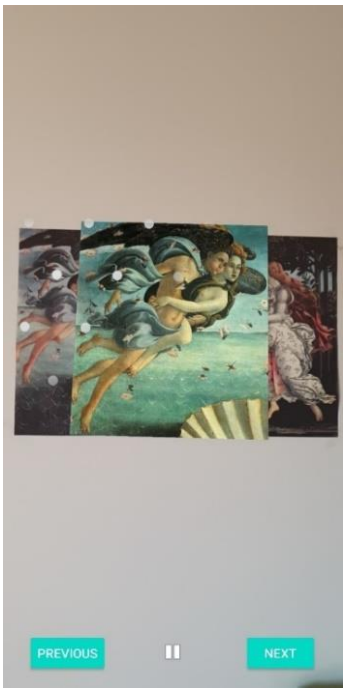


Figure 4.27 - Visual augmentation 2

4.5.2 PAINTING ANALYSIS

Here we provide the analysis of the painting used for the augmented storytelling. "The Birth of Venus", by Italian renaissance artist Sandro Botticelli was chosen as a case study for the experimentation of the application; such a piece lends itself perfectly to the purpose, given the rich history that characterizes it and the clear spatial separations of characters and elements in the scene depicted.

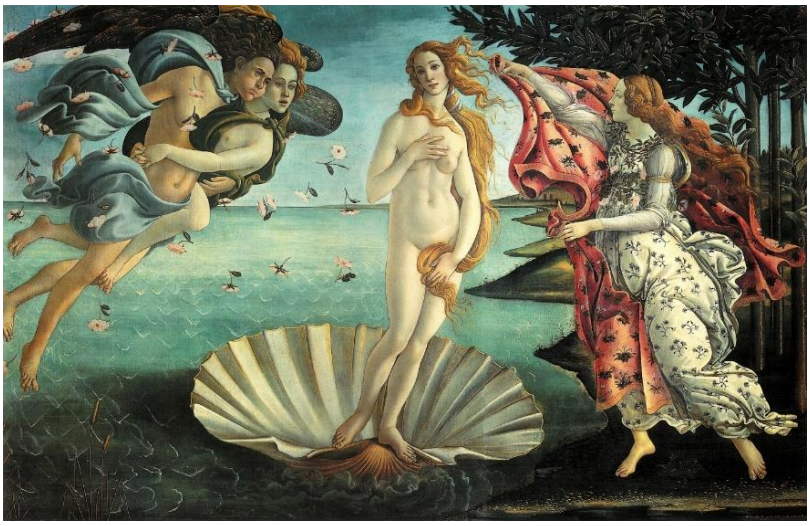


Figure 4.28 – The Birth of Venus by Sandro Botticelli

The following analysis, paraphrased and translated from Dario Mastromattei's study of the piece[10], has been divided into small units, on which the TTS storytelling has been built:

"*The main focus of the composition is the goddess of love and beauty, Venus, born by the sea spray and blown on the island of Cyprus by the winds, Zephyr and, perhaps, Aura. She is met by a young woman, sometimes identified as the Hora of Spring, who holds a cloak covered in flowers, ready to cover her. A detail often overlooked is the lack of shadows in the painting; according to some interpretations, the scene is set in an alternative reality, still very similar to our own.*"

"*The Goddess is standing on a giant scallop shell, as pure and perfect as a pearl. She covers her nakedness with long, blond hair, which has reflections of light from the fact it has been gilded. The fine modelling and white flesh colour give her the appearance of a statue, an impression fortified by her stance, which is very similar to the Venus Pudica, an ancient statue of the Greek-roman period.*"

"*You may wonder why Venus is standing on a shell; the story goes that the God Uranus had a son named Chronos, who overthrew his father and threw his genitals into the sea; this caused the water to be fertilised, and thus the goddess was born.*"

"*In the top left of the piece we can notice Zephyrus, God of the winds; he is holding Aura, personification of a light breeze. The two are highlighting the pale face of the Goddess, while blowing the shell towards the coast.*
Regarding Aura, some scholars are in doubt about her identity; she may in fact be Chloris a nymph which married Zephyrus in an alternative story."

"*The Hora herself may be a complementary version of the nymph Chloris. Are they two versions of the same person then? It might be; the story of this woman is narrated in "I Fasti" by Latin author Ovidio and the painted in "The Spring", by Botticelli himself, where the woman gets kidnapped by Zephyrus to become a mystical figure. The theory is quite farfetched, however there's a detail in its favour: the roses falling around her and Zephyrus.*"

4.5.3 DATA SCIENCE PROCESS

Following the schema of the Data Science Process from Chapter 1.1.1, we will be presenting the steps we followed during the construction of CiceroAR's training image dataset: firstly, the goal we set was obviously to recognize a particular painting in an image, with an acceptable accuracy; the data used to train the network came from a mixture of internal and external sources: we used part of a larger painting dataset from Kaggle.com, a website used for Machine Learning projects and competitions, and we snapped a series of picture containing the painting from a book and a poster in different lighting conditions and from different angles.
The images were prepared using a labelling tool, LabelImg, used to identify the regions of the images that contained the painting by drawing a rectangle around them; the tool then output an XML file for each one, containing the spatial coordinates. The only transformation applied to the images before the labelling process was a rescaling to a resolution of 800x600 pixels, in order to speed up the training process.

Here are two sample images from the small dataset used to train the neural network.

Figure 4.29 - Dataset sample 1

Figure 4.30 - Dataset sample 2

Example of an XML label file:

```
<annotation>
    <folder>test</folder>
    <filename>20200503_012225.jpg</filename>
    <path>
        C:\Users\Michelangelo\Desktop\Dataset\test\20200503_012225.jpg
    </path>
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>800</width>
        <height>600</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>Venere</name>
            <pose>Unspecified</pose>
            <truncated>0</truncated>
            <difficult>0</difficult>
            <bndbox>
                <xmin>205</xmin>
                <ymin>135</ymin>
                <xmax>599</xmax>
                <ymax>478</ymax>
            </bndbox>
    </object>
</annotation>
```

CONCLUSIONS

After testing the developed application on the selected painting, it was possible to verify the applicability of the used technologies, in particular relatively to the training mechanisms of the neural network, for which a limited number of samples produced a satisfactory result. Furthermore, the TTS storytelling was proven to be a valid component to expand the accessibility of the system, and the usage of AR techniques to accompany the narration revealed to be a fun and interactive way to perceive new information.

Although this work has been focused on the recognition of two-dimensional pictorial pieces and the narration of their history and details, with few modifications the system could be able to operate on three-dimensional artefacts, or it could even be generalized to recognized any kind of image, since the technologies fit this purpose quite good: an example could be a blueprint of a building on which, upon recognition, an Augmented Reality model of the final building could be placed.

The most noticeable difference when operating on 3D objects would be the method used to position the augmented elements in the scene: since the AugmentedImage module of ARCore, used to find the image in the scene after it has been recognized by the AI server, can only track 2D images, a different anchor point is necessary to place automatically the eventual images and models during the narration of a 3D piece. If, on the other hand, the augmentation would take place on the model itself, it would still be possible to manually place the virtual element to overlap its real-world counterpart.

When it comes to the neural network, a three-dimensional object would also make the training process considerably easier, assuming that it would be possible to create a dataset made up of images taken from different angles, an operation not quite possible with 2D images.

Finally, in order to further expand the accessibility and the target audience of the application, other narration options could be implemented, such as subtitles or some interaction mechanism between the user ad the virtual guide: this could be realized in the form of a real time smart chat or via a series of questions asked by the virtual guide regarding the narrated painting or object, both realized using speech recognition. Furthermore, the realized software, following an appropriate reengineering and adaptation process, could also be integrated as a service into other systems: a museum application, for example, could provide the use of the system following the purchase of a ticket or as a reference for specific targeted tours.

BIBLIOGRAPHY

[1] R. Szeliski, Computer Vision: Algorithms and Applications, Springer, 2010, page 3

[2] D. Kriesel, A Brief Introduction to Neural Networks, 2005, page 38

[3] D. Schmalstieg, T. Höllerer, Augmented Reality: Principles and Practices, 2016, page 3

[4] R.Azuma, A Survey of Augmented Reality, 1997

[5] P. Milgram, Augmented Reality: A class of displays on the reality-virtuality, 1994, page 2

[6] R. Azuma. Tracking requirements for augmented reality. Communications of the ACM, 1993

[7] S. Siltanen, Theory and applications of marker-based augmented reality, 2012, p40

[8] D. Cielen, A. D. B. Meysman, M. Ali, Introducing Data Science: Gib data, machine learning, and more, using Python tool, Manning, 2016, p. 233

[9] Fritz AI, Image Recognition Guide, https://www.fritz.ai/image-recognition/

[10] D.Mastromattei, La nascita di Venere di Sandro Botticelli: la filosofia che ha cambiato il mondo del ‘500, Arteworld, https://www.arteworld.it/la-nascita-di-venere-sandro-botticelli/