MASTER IN COMPUTER SCIENCE

Multimodality with HTML5

Mannard, John

Award date:
2015

Awarding institution:
University of Namur

SUMMARY AND KEYWORDS

ABSTRACT

Multimodality is a way of improving human-machine interaction. Studies have shown that it could enhance user productivity and working memory and that humans have a natural way of communicating that is multimodal. Since Bolt's "put-that-there" experiment in 1979, several multimodal frameworks and findings have emerged but none seem to be designed for WEB applications. Considering the predominance of WEB applications and the advances in WEB technology following the arrival of HTML5, multimodality support seems to be the next natural step for WEB applications. This thesis lists a set of existing technologies fit for multimodality, introduces a new WEB multimodal framework based on these technologies and finally illustrates its usage through a set of examples.

RÉSUMÉ

La multimodalité est un moyen d'améliorer les interactions homme-machine. Des études ont montré qu'elle pouvait accroître la productivité de l'utilisateur et sa mémoire de travail et que l'humain communique naturellement de façon multimodale. Depuis l'expérience du « put-that-there » de Bolt (littéralement, « mets ça là ») en 1979, plusieurs frameworks et travaux de recherches se sont intéressés au sujet mais aucun ne semble s'intéresser au domaine du WEB. Sachant que le WEB est aujourd'hui un domaine prédominant dans la technologie de l'informatique et que le HTML5 a fait émerger un ensemble de nouvelles technologies, il semblerait que la prochaine étape soit d'intégrer la multimodalité aux applications WEB. Sont citées dans ce mémoire les technologies existantes, puis proposé un nouveau framework multimodal orienté WEB illustré par un ensemble d'exemples.

KEYWORDS

Multimodal, WEB, EMMA, multimodal framework, put that there, speech, gesture, lattices, HTML5, canvas, WebRTC, WEB video, WEB audio, ink.

ACKNOWLEDGMENTS

I wish to express my sincere thanks to my girlfriend, who has supported and advised me from the very beginning. She has also agreed to proofread parts of this thesis, which has been of great help.

I would like to express special thanks to Bruno Dumas, my supervisor, who has escorted me throughout this whole thesis and has shown some remarkable patience and availability. I would like to thank him equally for the topic of this thesis, which has been quite interesting and has made me discover the whole field that is multimodality. I would finally thank his proofreading.

CONTENTS

GLOSSARY

Bottom-up approach: this term is used to describe a development process where the implementation of a library is gradually done by experimentations. The library is built by gradually adding small levels of abstraction to the raw technologies it is based on.

EMMA: W3C specification, stands for “Extensible MultiModal Annotation” language.

Fission: process in multimodal applications that aims to select a modality depending on the context.

Fusion: process in multimodal applications that aims to derive intelligence by fusing multimodality inputs.

Lattice: EMMA structure that represents or models a state machine of a set of possible utterances.

Modality: communication channel between the user and the machine.

OCR: stands for “Optical Character Recognition” and is an automatic conversion of scanned or handwritten texts into a string.

Put-that-there: classic example of multimodality where the user utters the “put that there” request to the machine while pointing at an object to move on the screen and pointing at a destination area.

Utterance: it can be both the act of speaking and a vocal expression

W3C: World Wide WEB Consortium. It is a major standardization agency for the WEB.

WIMP: acronym of Windows, Icons, Menus, Pointing Device. (Dumas, et al., 2009). WIMP applications are the most common that can be found currently regarding to human-machine interfaces.

INTRODUCTION

In human-machine interaction, communication happens through *modalities*. A modality is a communication channel, such as voice, gesture, body movement, touch, pen or even gaze. For instance, when the user speaks or listens to a machine, the *voice* modality is in use. Multimodality is an alternative class of human-machine interaction that tends to combine multiple modalities to offer a more natural way of interaction for the human (Oviatt, 2003). It is in competition with the classic WIMP applications (Window, Icons, Menus, Pointing device) (Dumas, et al., 2009). The “put that there” (Bolt, 1979) is quite known in multimodality and is often used to illustrate its concepts. In WIMP applications, moving an object across the screen is usually done by using the mouse. In multimodality, this same action can be performed by combining the voice and pointing modalities. The user utters the “put that there” command to the machine (oral modality), points at what object to move and where to move it (pointing modality).

Research in multimodality has been active for the past three decades, resulting in the release of multiple frameworks, languages, specifications and advanced recognition systems. Although some of them can show some very interesting results, it appears that most of them are designed for desktop applications only. There does not seem to be any web-oriented multimodal framework. In parallel, the arrival of HTML5 has been a major turnaround in web development. It gave the opportunity to manipulate advanced features like audio, video and drawing through the new “canvas” element. New technologies and research projects began to emerge as well, based on these HTML5 new features. Considering the current status of these technologies, this thesis aims to find out if multimodality can be integrated in web applications.

Chapter 1 gives a general introduction to multimodality. The goal is to draw the context of multimodality, by explaining the key principles and various technical terms that will be used further on. It also gives a rapid overview of the existing frameworks that are capable of managing multimodality and concludes by introducing the work that will be carried out in the rest of this thesis, which can be divided into three main parts.

The first part of this thesis is a research process. The goal is to build a list of modalities that could be implemented in web applications by using HTML5 new features. The W3C website is used as a starting point, as well as several documents about multimodality like (Dumas, et al., 2009).The EMMA language is interesting to consider as well, mostly because it is a web-oriented standard. The specification (W3C, 2009) is consulted as reference and summarized in chapter 5. The first step is to carry out intensive research on existing modalities-related technologies that could serve the purpose of this thesis. This includes finding the technologies and testing their capabilities as they come along. The testing is an important process to ensure that they are functional and that they do answer the needs of multimodality. Gradually, the technologies will be documented and evaluated. Chapters 2 to 4 show

the results of this research in detail. Voice, gesture and ink modalities are discussed respectively in chapters 2, 3 and 4. Chapter 6 is the last step of this first research part, which summarizes all findings into one table and builds a review about them. Depending on the results of this review, every library presented in chapters 2, 3 and 4 will be declared as suitable or abandoned.

The second part of the thesis is led by the purpose of building a multimodal framework for web applications by using all technologies previously found. The first step will be to design a global architecture. Due to inherent uncertainties of multimodality and the used technologies, building the multimodal framework requires a bottom-up process (which is justified in details in chapter 7), where the framework is gradually built in parallel to the examples of chapter 8. These examples equally serve as validators of this framework and illustrate its capabilities. They are based on classic features of multimodality.

The final part is an assessment process, in which a step back will be taken to evaluate the work of the thesis. A review of the framework developed through the second part of this thesis is discussed in section 9.1. This aims to lay down clearly all the key points and weak points of this framework based on actual implementation and choices of the used technology. A set of future directions will be given in section 9.3. These future directions aim to explain how the framework can be extended and enhanced by considering the strengths and weaknesses of the framework illustrated in section 9.1. Finally, an assessment of the EMMA language will be presented in section 9.2. This aims to evaluate its adequacy to multimodality in retrospect of its practical usage across this thesis.

1. STATE OF ART

1.1. MULTIMODALITY

Multimodality literally means « multiple modalities ». In human-machine interaction, a modality is a communication channel between two entities. Given the fact that a human being can hear and speak, voice is a modality, just like gesture, body movement, touch, pen and gaze. Multimodal applications tend to include multiple modalities and infer actions from them, which is much closer to the human natural way of communicating. Multimodality emerged about 35 years ago with Richard Bolt (Bolt, 1980) who created a revolutionary application type that he simply called “put that there”. This example is discussed further in section 8.5. The application could draw and move basic shapes across the screen, simply by listening to the user speaking and pointing. Since then, gesture and voice have been the predominant modalities researched. Research in these topics have happened both in psychology where was studied the cognitive properties of human interaction and in technical and engineering domains where researches came up with technical solutions. It has been proven that multimodal applications tend to enhance human interaction with the machine and by extension user performance (Oviatt, 2003). Indeed, cognitive psychology research has shown that due to human perception, communication and memory function, injecting multiple modalities tends to increase human working memory, which increases performance (Oviatt, 2003). For instance, a speech recognizer modality would allow the user to speak a command while being busy on another task, such as writing on paper. This would be possible only in a multimodal application and would have to be executed in sequence in front of a classic WIMP (Window, Icon, Menu, Pointing device) application. Hence, multimodality is an elegant way of resolving issues of handicapped people. Blindness is for instance a big issue when it comes to WIMP applications. Multimodal applications have the capabilities of allowing plural ways of communication, from which the user can make a choice.

Figure 1.1 illustrates how communication can be modelled. A human always does an action, which can be either voluntary (speaking to the machine) or involuntary (facial expression, blinking). The multimodal application captures the event through its sensors and then proceeds to a fusion process which aims to combine all events that can be grouped together as part of the same intent. Afterwards, the fission process happens, where the system will decide how and through which modality the user will get the result, depending on the context.

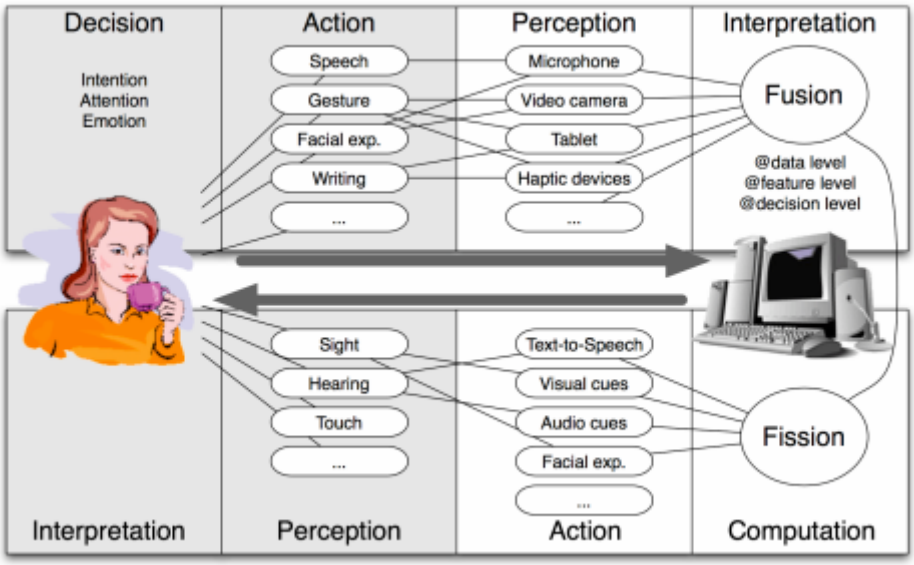


Figure 1.1 Multimodal communication model (Dumas, et al., 2009)

A multimodal system that could be integrated in a communication system like illustrated in Figure 1.1 should possess a set of components as represented in Figure 1.2. Input modalities are perceived through the recognizers that then send their data to the fusion engine. The fusion engine's responsibility is to extract intelligence from a set of input modalities. Various levels of fusion are possible: data-level fusion, feature-level fusion and decision-level fusion (Dumas, et al., 2009). Data-level fusion happens when inputs passing through a modality are coming from multiple sources, like two web cameras filming in parallel. Feature-level fusion is used when two modalities that are tightly coupled have to be fused because they are part of the same intent. To manage loosely coupled modalities, the decision-level fusion can be used.

The dialog manager, getting the interpretation from the fusion engine, is responsible for determining the appropriate response. There are several approaches to achieve dialog management (Dumas, et al., 2009): finite-state approaches, information state-based approaches, plan-based approaches and collaborative agents-base approaches.

The fission engine is finally in charge of deciding *how* the response will be addressed to the user. This response can happen through a single modality or through multiple modalities, in which case a fission process will be necessary. The response's format highly depends on the context in which the user is using the application. This is the responsibility of the context user model history which keeps track of this information.

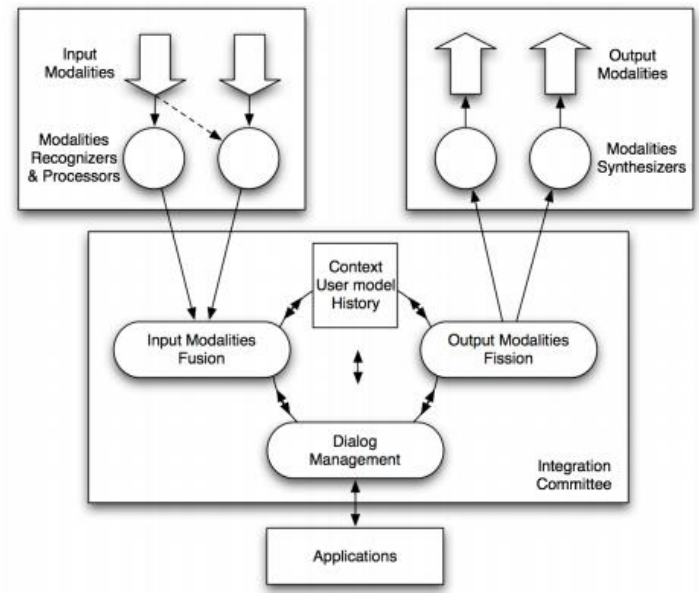


Figure 1.2. The architecture of a multimodal system (Dumas, et al., 2009)

Multimodal applications are much more complex to prototype and to model than classic graphical user interfaces and WIMP applications. Indeed, the fusion and fission processes require some deeper thinking than simply a click on a button. Machines are not built to fit human natural interaction naturally. It is therefore useful to be able to model multimodal applications. The CARE model (Dumas, et al., 2009) focuses on how modalities can be combined at the user-machine interaction level. CARE is the acronym of "Complementarity, assignment, redundancy and equivalence", which are four properties introduced by the model. Complementarity describes a system where multiple modalities have to be interpreted as part of the same intent to be meaningful. Bolt's "put that there" experiment perfectly illustrates complementarity of pointing and voice modalities. Assignment describes the fact that only one modality can be used for a particular action. For example, cancellation of all unsaved changes should only be performed through a clearly voluntary modality, such as touch. Redundancy is used to perform the same action through different modalities. For example, a waving gesture and a "goodbye" utterance can be fused as the same "shutdown" command but through different modalities, which are gesture and speech. Finally, equivalence is a way of multiplying techniques to perform the same action. For instance, the user could control a video by clicking on standard control buttons or by speaking the commands.

1.2. EXISTING TECHNOLOGY

There currently exists a set of frameworks handling multimodality. According to the work of (Cuenca, et al., 2014), these frameworks (or toolkits) can be classified into three groups: flow-based, state-based and token-based. Flow-based toolkits have models that describe only the flow of data that is coming from the end user. They do not offer support for fusion and fission processes as well as dialog management.

State-based toolkits were named because their models looked like state machines. These include a fusion engine and a dialog manager. Finally, token-based toolkits were named because of their use of the “token” symbol for the modelling of parallelism. Table 1.1 lists all the different frameworks and in which group they are classified:

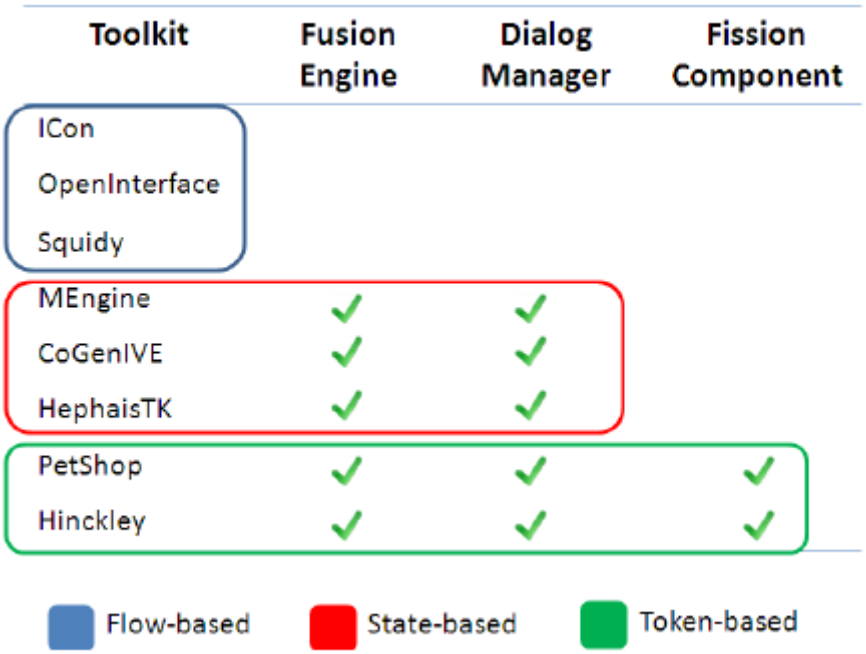


Table 1.1. Comparison of existing frameworks and their classification

ICon is a framework written in Java that focuses on the ease of plugging hardware devices to an application. In other words, ICon offers support for a wide and very extensible set of heterogeneous input devices that can be added to any Java application. The set of input devices can be easily extended thanks to the framework’s architecture. ICon does not offer support for fusion, fission and dialog management, which is left to the programmer.

Squidy (König, et al., 2010), an integrated tool coupled with a GUI, allows creating multimodal interfaces through graphs of data flow. The flow diagrams are directed graphs where nodes represent hardware devices or filters and pipelines describe the flow of information. It includes various input modalities such as multi-touch input, pen interaction, speech recognition, laser pointer and gesture tracking. It is written in Java.

OpenInterface (Serrano, et al., 2008) is another multimodal framework and is similar to Squidy. It was the result of a European research project. Its SKEMMI graphical editor allows creating multimodal systems by creating directed graphs. In these graphs, nodes represent autonomous components of the system that can be written in any language, as long as their interfaces are described in a XML-based language

called CIDL. The arcs of the directed graph aim to connect the outputs of one component to the inputs of another. Some components are pre-defined (the OI Adapters) and offer common features such as filters, buffers and data transformers.

MEngine (Bourguet, 2002) is a state-based toolkit that abstracts a multimodal application as a state machine. As opposed to flow-based toolkits, the state-based toolkits include a fusion engine and dialog manager. Thanks to a component named IMBuilder, the user can define a set of state machines that describe legal interaction with the machine. The MEngine, upon each event arrival, checks for all state machines if there is a match of a composite interaction. If there is, it notifies the client application that a match has happened and the subroutine to execute.

CoGenIVE (Cuenca Lucero, 2013) is a multimodal toolkit that is used in combination with the NiMMiT (Vanacken, et al., 2006) modeling language. It receives NiMMiT as input, which describes techniques of interaction. A NiMMiT model is a state machine containing a set of specialized symbols:

- Circle: state of the system
- Arc: user event
- Rectangle: task to be executed by the system
- Label: information to be stored or retrieved by the system

Obviously, NiMMiT gives the opportunity to compose events to create a multimodal system.

HephaisTK (Dumas, et al., 2009) is a toolkit for multimodality focusing on events and fusion algorithms. It uses the SMUIML description language (Dumas, et al., 2010). An example of this language is illustrated in Figure 1.2. There is a graphical editor to edit SMUIML by drawing graphs. The ellipses represent the states of the system, while the arcs abstract the transitions between those states. These arcs can be annotated with the triggering event and the action to perform during the transition. The events can be composed by including them into the same container. Four types of containers are available:

- Seq-And: includes a set of complementary events that have to be triggered sequentially.
- Par-And: includes a set of complementary events that can be triggered in any order.
- Seq-Or: includes a set of equivalent events that have to be triggered sequentially.
- Par-Or: includes a set of equivalent events that can be triggered in any order.

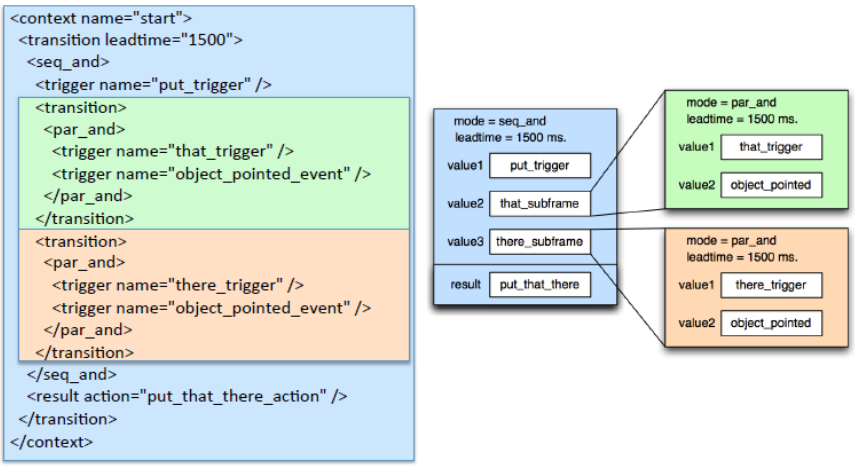


Figure 1.2. Example of SMUIML description language

Petshop (Navarre, et al., 2009) is a graphical toolkit that is capable of editing an ICO (Interactive Cooperative Objects) formalism. The ICO concept is an interactive system where the components communicate by responding to the services they offer and making requests to the other components. Basically, the flow of information happens through what they call in (Navarre, et al., 2009) *tokens* that can move across the graph and duplicate. The presence of these tokens in the different nodes can activate or not transitions, depending on the rules that were defined.

The Hinckley's language (Hinckley, et al., 1998) has no graphical toolkit available. However, it can be abstractly visualized as a directed graph with circular nodes, arcs and tokens. The purpose of this language was to represent interaction techniques using two hands. Therefore, the circular nodes can be dashed or solid depending on the hand that was used to trigger the event. Each hand holds a device, represented on the graph by a gray and black token. This technique allows to browse the graph in parallel and to fuse different actions into one, such as zooming out that can be performed by simultaneously moving the two devices apart.

Mudra (Hoste, et al., 2011) is another framework using a declarative rule language to build multimodal scenarios that is not referenced in table 1.1. A fact base is built using this rule language and the RETE algorithm (Forgy, 1982) is used for interpretation. An example of usage is shown in Figure 1.3, which illustrates the classic "put that there" example. It simply define "put that there" as a sequence of three words happening simultaneously with two acts of pointing. The first one has to happen within 400 milliseconds of the "that" utterance, while the second one has to happen within 400 milliseconds of the "there" utterance.

```
rule :bolt do
  put = Speech.word “put”
  that = Speech.word “that”
  p1 = Point
  Parallel [that, p1] { :time <= 400.ms }
  there = Speech.word “there”
  p2 = Point
  Parallel [there, p2] { :time <= 400.ms }
  Sequence [put, that, there]
end
```

Figure 1.3. Example of rule in the Mudra framework

Despite those multiple existing technologies that simplify the creation of multimodal applications, there does not seem to be any web-oriented framework allowing the building of multimodal web applications. The HTML5 specification rise has brought a considerable amount of features to web development to increase its dynamics. For instance, the WebRTC standard (W3C, 2015) has brought camera and audio input to the browser, meaning that data can be retrieved from those devices and used for web applications. With those new features came a set of libraries abstracting the complexity of recognizers, which will be discussed in the next chapters. Moreover, W3C has issued a number of specifications that were followed by some browsers: WebRTC (W3C, 2015), multimodal interactions (W3C, 2014), media capture (W3C, 2015) and audio capture (W3C, 2012). Although most browsers do not have support yet for audio and video input, it is much likely that in the next few years they will all evolve and offer support for these topics. In the light of these major advances in web development, it is time to introduce multimodality in web browsers. This will be the goal of this thesis, after some research on the existing technologies on the subject of recognizers. A library abstracting and facilitating the creation of multimodal web applications will be presented in chapter 7 and will be tested through a list of examples in chapter 8.

2. VOICE WITH HTML5

2.1. INTRODUCTION

When thinking of modalities, speech is probably the first one that comes to mind. Its main advantage is that it is faster for the user. For instance, journalists often use recorders for their interviews because speaking is a much faster process than writing it.

Technically speaking, voice can be both an input and an output. When the user is speaking to the machine using words of a natural language, it is considered as an input and is called speech recognition. In this process, the machine will physically record the user speaking and convert it into text, which can then be understood by a program. When the machine is speaking to the user, this is an output and is called speech synthesis. Technologies do exist in voice recognition and synthesis. In this chapter will be presented the *webkitSpeechRecognition* and *speechSynthesis* APIs available for web development.

2.2. SPEECH RECOGNITION

In Google Chrome 25 appeared a JavaScript based speech recognition tool called *webkitSpeechRecognition* (Google, n.d.). Basic usage is as follows (Ornbo, 2014):

```
var rec = new webkitSpeechRecognition();

rec.onresult = function(event) {

    // event management

}

rec.start();
```

This tool can be used in several ways. There are lots of options that can be set by the programmer, granting them freedom of use.
Some examples of usage:

- Continuous listening on a webpage of commands spoken by the user (browsing to a webpage, retrieving information, process operation in background). This API can for example be used to control a video player (Devlin, 2014).
- Transcription of a text spoken by the user in a text field or text area.
- Asking a question to the website.
- Automatically recording a meeting or conversation.
- Multimodality
- Resolving some disabled people problems in interacting with the web page.

WebkitSpeechRecognition is based on google API and requires an internet connection. The database used for speech recognition can be quite substantial and

thus cannot be held on the client's side. It was built with the idea to improve the HTML5 x-webkit-speech component, which was not quite flexible and controllable by programmers (Google, n.d.). Even if basic usage was easy, it still required scripting to implement advanced recognition systems. All processing and decision making were hidden, and not controllable by programmers.

WebkitSpeechRecognition is an implementation of the W3C specification interface (W3C, 2012). The interface of this specification is included in appendix 1. In the following sections, the stress will be put on some elements of this specification and how it was implemented in the WebkitSpeechRecognition API. The list is not exhaustive and can be fully consulted in the W3C specification.

2.2.1. LANGUAGE

There is an option for setting the language. Obviously, this is an important feature, as it is simpler to recognize one language than several ones that could be combined together. The Google API implements this feature. It is able to recognize multiple languages, but is slightly faster when the language is set. It is still able to detect and recognize words in other languages though, even if the language property is set to another language.

Examples:

```
recognition.lang = 'fr-FR';

recognition.lang = 'en-GB';
```

2.2.2. INTERIM RESULTS

Interim results are defined by the specification as temporary results, allowing a web application to do real-time speech recognition. This property is set to false by default, meaning that the speech recognizer tool will wait for the user to finish his speech before analyzing the sentence. Interim results are recognized in real-time, giving the impression of direct transcription of the words spoke by the user.

```
recognition.interimResults = true;
```

2.2.3. CONTINUITY

The continuous parameter determines whether the speech recognizer must continue processing voice input or stop after having processed the first words. When set to

true, speech recognition continues until the service is manually stopped (by the stop() method).

```
recognition.continuous = true;
```

2.2.4. METHODS

There are three methods in the speech recognition class.

- start(): starts recognizing speech, and opens the connection if it has not been done already.
- stop(): stops recognizing speech at once, but returns remaining results if any.
- abort(): stops recognizing speech abruptly, without further actions (unlike stop() that returns the remaining results if any).

2.2.5. EVENTS

- onaudiostart(): user agent starts to capture audio
- onaudioend(): user agent stops to capture audio
- onsoundstart(): sound has been detected (possibly speech)
- onsoundend(): sound has ended
- onstartevent(): user agents start recognizing
- onstopevent(): user agent has stopped

It must be noted that the onsoundstart and onsoundend events are not working properly in the Google specification. It only works if the continuous property has been set to false. There is therefore no way for the programmer to know when the user is speaking, or at least completely. For example, if the webpage is continuously listening to the user, the soundstart event will be launched only once, when the user speaks for the first time, and the soundend event launched once when the speech recognition ends. As a result, if the user decides to start talking, then stops, and resumes seconds later, there is no way to differentiating the moments of silence from speaking.

2.2.6. RESULTS

The results can be processed by the programmer by implementing the onresult event.

```
recognition.onresult = function(event) {

    console.log(event);

}
```

The understanding of the event object received is complex due to its nested values.

The most interesting element is the **list of *speech input results***. This list can contain several elements if used in a continuous process, but always contains one single result if the process is not. The result can be final or not. It is final when it is not an interim result.

A ***speech input result*** contains a ***list of speech input alternatives.*** This list contains a list of interpretations containing the actual results and a confidence value between 0 and 1. The maximum number of alternatives can be defined by the property *maxAlternatives* set by default to 1.

2.2.7. EMMA GENERATION

The W3C specification gives the interface for programmers to retrieve the produced EMMA (as described in chapter 5) document in the event object. However, it does not seem to have been implemented in the Google tool yet. EMMA generation is thus left to the programmer for the moment.

2.2.8. WORKING EXAMPLE

```
var rec = new webkitSpeechRecognition();
rec.continuous = true;
rec.maxAlternatives = 1;
rec.interimResults = true;
rec.lang = 'en-GB';
rec.onresult = function(event) {
    var interim_transcript = '';
    for (var i = event.resultIndex; i < event.results.length; ++i)
    {
        if (event.results[i].isFinal)
            final_transcript+= event.results[i][0].transcript;
        else
            interim_transcript+=
              event.results[i][0].transcript;
    }
    $('#final_span').html(final_transcript);
    $('#interim_span').html(interim_transcript);
  }
rec.start();
```

```
<div id="text ">
    <span class="final" id="final_span"></span>
    <span class="interim" id="interim_span"></span>
</div>
```

When used in a web application, this example will copy in real-time the text that the user is currently speaking into the “span” element.

The tool is able to translate punctuation as well, such as “comma”, “dot”, “exclamation mark”. When the user utters a punctuation mark, the API is capable of directly translating into the corresponding graphical punctuation mark.

For instance, uttering “My name is John Doe dot I will be your user today exclamation mark” will produce the result shown in Figure 2.1.

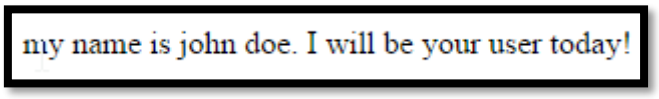


Figure 2.1. Result example of the speech recognition API

2.2.9. DISCUSSION

The *WebkitSpeechRecognition* API is quite powerful in some aspects. It is able to process results and update them after having considered the rest of the sentence. It has a good flexibility for programmers. It is handy to use and follows a W3C specification.

Nevertheless, as it is implemented as an API, it needs processing over the network, which can drastically slow down the tool. It is not fully functional, as it has trouble recognizing certain sentences, and it requires a proper accent with no language flaws. It is thus not fit for web applications that are used in a real context by unaware or untrained end users. The API is as its best when recognizing small sentences or words. For example, it could be used in a web application for browsing different pages by speaking the name of a page. However, this would still require the user to choose his words carefully and speak slowly with a proper accent.

WebkitSpeechRecognition requires user approval before starting but does not need it on a secure HTTP website (https). This could be seen as a violation of privacy by users because it means that every word spoken could be collected by the website. Moreover, once started, if the web application does not have an explicit “stop voice recording” feature, there is no way for the user to stop it. The user may give their consent at first, but needs to close the webpage to stop the web application from listening.

Finally, Google Chrome is for now the only browser to have implemented speech recognition. The following code can therefore be used to check browser support:

```
if ('webkitSpeechRecognition' in window) {

    ...

}
```

```
else {
    alert('No speech recognition!');
}
```

2.3. SPEECH SYNTHESIS

As said in the introduction, speech synthesis is the computer speaking using an understandable natural language. W3C has worked on a specification that can be fully consulted on their official website (W3C, 2012). The specification for speech synthesis can also be consulted in appendix 2.

Google Chrome and Opera browsers have both implemented this interface and can be used freely. Speech synthesis is not yet supported in Internet Explorer, Mozilla Firefox or Safari.

Speech synthesis is another step into voice-driven web applications. It gives the programmer the ability to make a web application speak, which can be intrusive but is a good solution to get the user's attention. It can be used for different goals:

- Notifications to the user over voice
- Better interaction with people having sight problems
- User can hear a text to check style instead of reading it.
- Multitasking (hearing a text while browsing another tab).
- …

In the next subsection will be explained how this technology can be used in real web applications. Tests have been conducted in both Opera and Google Chrome.

2.3.1. METHODS

The JavaScript class has a bunch of methods:

- speak: starts speaking the text of the utterance object
- pause: pauses text for later resume
- resume: resumes text that has been paused
- cancel: cancels speaking with no chance of resuming
- getVoices: returns a list of available voices. The list returned is empty though in both Chrome and Opera.

2.3.2. THE UTTERANCE OBJECT

When approaching speech synthesis, one must understand the concept of utterance. An utterance is by definition both the act of speaking and a vocal expression. It is therefore both the action and the result. In the speech synthesis object, an utterance

is only an input that has to be spoken. It is represented by the *SpeechSynthesisUtterance* class, which has several properties:

- text: what has to be spoken
- lang: the language in which to speak. Opera does not support it.
- volume: volume at which to speak. Value must be between 0 and 1, and is 1 by default. The property does not seem to be working in Chrome.
- rate: the rate at which to talk. 1 is the normal rate, above is faster and below is slower. It does not seem to be working in Chrome.
- pitch: pitch of the voice. Default is 1, can be 0 and 2. It does not seem to be working in Chrome.

2.3.3. THE UTTERANCE EVENTS

The speech synthesis utterance object has those explicit events: onstart, onend, onerror, onpause, onresume.

It has another particular event: onboundary. This event is fired when a word or sentence is starting. The W3C Specification actually says it should be fired when reaching a “boundary” but the Opera implementation seems to fire it at the beginning of the word or sentence. The name attribute of event tells if it is a “sentence” or a “word”. Events are not fired in Google Chrome.

According to specification, fired event objects should have useful information like the char index that had been reached when the event was fired, as well as the elapsed time since the beginning of the utterance. However, this is not implemented in both Chrome and Opera.

2.3.4. WORKING EXAMPLE

In the following example, the user can type a text in an HTML text area and, when clicking “Please read”, will be converted into an utterance.

```
$(document).ready(function(){
    $('#read_me').click(function(){
        var msg = new SpeechSynthesisUtterance($('#words').val());
        msg.lang = 'en-GB';
        window.speechSynthesis.speak(msg);
    });
    $('#stop_me').click(function(){
       window.speechSynthesis.cancel();
    });
    $('#pause_me').click(function(){
       window.speechSynthesis.pause();
    });
    $('#resume_me').click(function(){
       window.speechSynthesis.resume();
    });

});
```

```
<html>
    <head>
        <title>Speaker</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <script src="../libraries/jquery.js"
                type="text/javascript"></script>
        <script src="speaker.js" type="text/javascript"></script>
    </head>
    <body>
        <div>
            <label>Type what you want to be spoken</label><br/>
            <textarea id='words' rows="5" cols="90"></textarea>
            <br/>
            <button id='read_me'>Please read</button>
            <button id='stop_me'>Stop now</button>
            <button id='pause_me'>Take a break</button>
            <button id='resume_me'>Break's over!</button>
        </div>
    </body>
</html>
```

2.3.5. DISCUSSION

As in speech recognition, speech synthesis is handy to use for a programmer. The specification clears the path for powerful voice-driven web applications, but implementation is still not at its best. Opera has implemented the most features, Chrome being interesting only for language support. Chrome lacks experience and robustness. Indeed, it has happened during the tests that the API randomly stops responding after a moment, due to either a buggy internet connection or the API itself. Speech synthesis object is also instantiated by the browser, which is problematic. If a text that was being spoken is paused by the user, who then reloads the web page to speak the whole text again, nothing will happen. The text will have to be resumed, with the utterance resuming at the exact point it was paused, and then canceled. A programmable instantiation would be more useful and would give more control to the programmer.

Finally, as said above, speech synthesis is only available in Google Chrome and Opera. According to (Bidelman, 2014) it is also partially supported in Safari iOS7.

Browser support of speech synthesis can be tested with this bit of code:

```
if ('speechSynthesis' in window) {

    ...

}
```

```
else {
    alert('No speech synthesis!');
}
```

2.4. PERFORMANCE

Performance of the webkitSpeechRecognition API is globally operational but still not fit for business applications. It requires some patience, forgiveness to errors and could lead to frustration when put in front of an end user. The API sends data to a google server for interpretation, explaining part of its slowness, but not all of it. In fact, this API has two types of results: final and interim results. As the user speaks, interim results are computed but are most of the time incorrect or redundant. They are however rapidly computed, as opposed to final results, and are a way of accelerating the web application. Nevertheless, one needs to take a step back when examining those results. Indeed, each time an utterance is heard, there are several interim results generated, sometimes incorrect, sometimes correct but redundant. One could therefore interpret several utterance results as the user repeating the same utterance while, in reality, it was only spoken once. Final results are more reliable but not enough. Moreover, there is always a given timeout before the API considers that the user stopped talking. That means that it adds a considerable delay to the network communication and the interpretation process.

Finally, there is the problem of interferences between the speech synthesizer and the speech recognizer. Indeed, the speech recognition API recognizes the synthetized speech. Therefore, if no actions are taken, a web application containing both will risk to be caught in a loop. For example, if the user speaks an invalid request, which is handled by the application by uttering “invalid request” through the speech synthesizer and this utterance is then taken account by the speech recognizer, which is a valid request at its turn, the web application will result in an infinite loop. Moreover, as the speech synthesizer is instantiated by the browser, it is impossible to stop all queued utterances by other means than closing the browser itself.

Rigorous tests about performance were not conducted in this thesis. However, if some should be made, it would require to measure two parameters: delay and reliability. The delay is the measurement of time elapsed between the user is done talking and the API giving a text result. Reliability could be measured by conducting tests on a big set of utterances with different accents, speech rates, pitches, volumes and surrounding noise. These tests would then lead to statistic results showing reliability. Finally, it is recommended to conduct tests on three types of systems. The first system would have to take into account only final results, the second one would take into account interim results and the third one would be a hybrid system filtering interim results based on some criteria. These criteria can be based on the confidence parameter and an algorithm managing duplicates.

3. GESTURE WITH HTML5

3.1. INTRODUCTION

If voice can be both an input and output, gesture is rarely an output in machines. When narrowing to web browsers with HTML5, it is even less common but could exist. There could be for instance an animated character on the web page waving, smiling, and running across the screen. Some animated smileys could be considered as gesture outputs. Nevertheless, this is a much more complex issue requiring drawing and image processing skills and will not be addressed in this thesis.

Gesturing is for humans a natural process. It is less natural to machines to understand. Despite the fact that modern image processing is able to recognize emotion and general body language, classic applications do not generally take into account gesture as data, while this data can sometimes be really helpful. Gesture is often a way to emphasize language when it comes to human conversation. A whole sentence can be interpreted as a joke or as an insult depending on *how* the person said it, with appropriate gesture or not.

In this next section will be discussed a set of technologies able to recognize gesture in HTML5 web pages. Touch will first be discussed as it can be very helpful and is a wide-spread technology nowadays. Then will be presented gesture and how the user can be understood through this modality within a HTML5 web page.

3.2. TOUCH

Touch is the action of touching a screen. This action can be faked on non-touchable devices with a mouse click. The benefit of this solution is that it has been around for years in JavaScript and frameworks like JQuery, which makes it trustful and handy to use. There is a lot of documentation on how it works. Touch handling can be achieved using events, which are listed here below:

- Mouse events: onclick, onmouseover, onmouseout (can be considered as touch on touch devices)
- Dragging events: ondrag, ondrop, …
- Touch events on touch screens: ontouchstart, ontouchmove, ontouchend, ontouchcancel

Events received as parameters of these functions know the coordinates of where the click happened. There is JavaScript function called *elementFromPoint(X, Y)* capable of returning an element of the web page by its coordinates. This is however not always accurate. Therefore, it is probably safer to use ids or classes (HTML5 attributes) to know for sure which element is being dealt with.

3.3. POINTING

Pointing is probably one of the most difficult processes to analyze. Even in real life, it is inaccurate. If the pointed object is far away and quite small, it can be hard for a person to show it to another person, as perspectives are not the same.

In software, it is often not achievable due to a lack of hardware. Pointing needs a three-dimensional space – so at least two cameras – whereas devices often deal with two-dimensional space only. It can however be "faked" by placing an indicator on the screen. This is what is done by the Kinect for instance. It displays a hand on the screen indicating a "start point" from where it will follows the human user's real hand. This gives some feedback on the human user that can then adapt and communicate with the machine.

Pointing will be discussed further in chapter 8 as there is a whole example built on it.

3.4. GESTURE

Gesture is a human way of emphasizing speech or any type of communication. It can relate to any type of physical movement of the body, including face expressions, body language, posture, waving…

In this section will be discussed what currently exists in terms of gesture analysis and video capture.

3.4.1. WEBRTC

WebRTC is a free and open source project launched by Google, followed by Mozilla, Opera and W3C (W3C, 2015) with the goal to develop real-time communications over the web using simple APIs. WebRTC is particularly meaning to enhance use of video and audio over the web, as well as offering support for RTP (Real-Time Protocol) communications.

WebRTC comes with three main APIs:

- *GetUserMedia*: Camera, microphone or screen of the device.
- *PeerConnection*: high-level media communication with abstraction of encryption.
- *DataChannel*: possibility to interact with other browsers directly.

Figure 3.1 is a diagram of WebRTC architecture. Browsers are responsible for the C++ implementation and offer to web programmers simple APIS for use. Audio capture has already been addressed in chapter 2. The video engine will be addressed in this section. Transport will not be addressed at all in this thesis as it is not part of its subject.

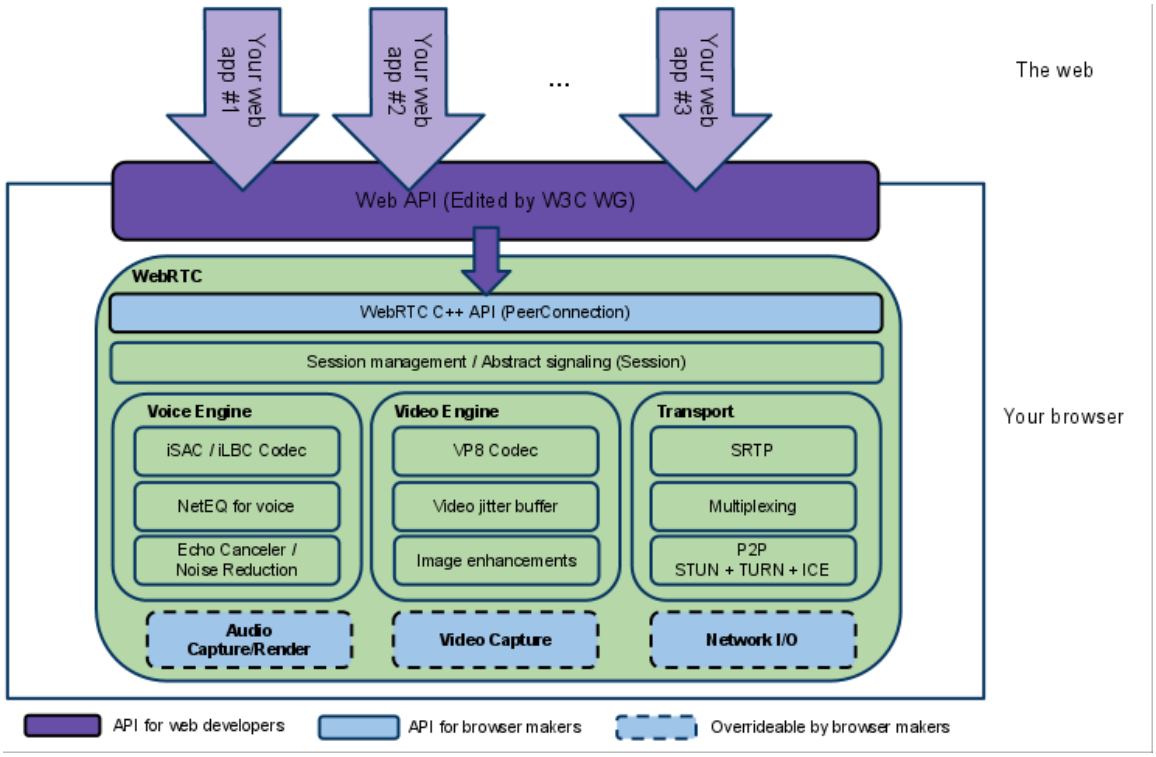


Figure 3.1. WebRTC architecture (WebRTC, 2011)

W3C specifies the WebRTC component (implemented in the browser) and the Web API component (W3C, 2015). We will below focus on the Web API specification.

The main method to remember is the getUserMedia one, asking the user authorization for video or input. This method takes three parameters:

- *Constraints*: dictionary with two keys: audio and video. Those are booleans requesting or not the use of audio/video.
- *Success callback method*: called when the user media has been properly initialized and is ready to be used.
- *Error callback method*: called when a problem has occurred.

Even if this method is standardized, it does not have the same name in every browser. Currently, only Opera, Chrome and Firefox implement the WebRTC APIs, and they all have different names: *getUserMedia*, *mozGetUserMedia* and *webkitGetUserMedia*. This difference and incompatibility can be fixed by using the following JavaScript statement:

```
var getUserMedia =
    window.navigator.getUserMedia ||
    window.navigator.mozGetUserMedia ||
    window.navigator.webkitGetUserMedia
```

3.4.2. HTML5 MEDIA CAPTURE EXAMPLE

It is possible to capture the web camera output and to render it directly on the screen. This can be used for example in WEB communications over webcam. It can also be used as feedback. Feedback is often a pleasant feature to have in web applications that use the camera input, even more when the web application is not reliable. It gives some indications to the user about what is happening behind, which allow them to adapt their behavior. This feature is even more useful for debugging a web application that uses the camera input.

The code below starts the video web camera and renders it into a web page (Bidelman, 2012).

```
<video autoplay id="video"></video>


<script>
    // Error function in case user refuses to allow camera
    var errorCallback = function(e) {
        console.log('User rejected webcam', e);
    };


    // For compatibility between browsers.
    navigator.getUserMedia  = navigator.getUserMedia ||
                    navigator.webkitGetUserMedia ||
                    navigator.mozGetUserMedia ||
                    navigator.msGetUserMedia;


    // Gets video element
    var video = document.querySelector('video');


    if (navigator.getUserMedia) {
        navigator.getUserMedia({audio: false, video: true},
          function(stream) {
            // Connect to video stream
            // Creates an URL blob from video stream and assign it to video
            video.src = window.URL.createObjectURL(stream);
```

```
        }, errorCallback);
    } else {
        video.src = 'somevideo.webm'; // fallback.
    }

</script>
```

Video capture is only the first step when building web applications around gesture. It can be very useful now to analyze this input, locate elements on the screen like hands, faces… and finally analyze movement to generate appropriate response.

3.4.3. MOTION DETECTION WITH JS-OBJECTDETECT

There is a motion detection JavaScript library (Mtschirs, 2015) based on the work of Paul Viola and Michael Jones of a face detection algorithm (Viola & Jones, 2004) using Haar wavelets (Haar, 1911). It has the ability to detect elements of the human body using the web camera and by defining “classifiers” (arrays of data) that allow the core algorithm to locate the elements. Here is an exhaustive list for which the classifiers are already defined:

- Eye
- Frontal face
- Full body
- Hand fist
- Open hand
- Mouth
- Nose
- Profile face
- Smile
- Upper body

The default use of this library is to define a *callback* function that is called by the browser each time we request an animation: *requestAnimationFrame(callback).* An alternative way of doing this would be to use the JavaScript *setInterval* function that will call the script at a regular interval. This solution is useful when the processor is too slow and is easily overloaded.

The library offers a compatibility class for compatibility issues between browsers. In fact, functions like *requestAnimationFrame* or *getUserMedia* for the webcam have different names in different browsers. The compatibility class abstracts all of this for the programmer.

In the callback functions, the library detector can be called to detect the desired object. If any result is found, the library returns an array of four coordinates which describe a rectangle wrapping the object on the screen.

In the following example is illustrated fist detection:

```
$(document).ready(function(){

    /*
     * Prepare canvas with video
     */
    video = document.createElement('video'),
    fist_pos_old,
    detector;

    /**
     * Prepare video webcamera
     */
    try {
        compatibility.getUserMedia({video: true}, function(stream) {
                try {
                        video.src =
                           compatibility.URL.createObjectURL(stream);
                } catch (error) {
                        video.src = stream;
                }
                // Listener to frame animations
                compatibility.requestAnimationFrame(play);
        }, function (error) {
                alert("WebRTC not available");
        });
    } catch (error) {
        alert(error);
    }

    function play() {
        // Continue listening
        compatibility.requestAnimationFrame(play);

        // Make sure video is ready
        if (video.paused) video.play();

        if (video.readyState === video.HAVE_ENOUGH_DATA
                                        && video.videoWidth > 0)
        {
            /* Prepare the detector once the video dimensions are known: */
            if (!detector)
            {
                var width = (200 * video.videoWidth / video.videoHeight);
                var height = 200;
                detector = new objectdetect.detector(width, height, 1.1,
                                        objectdetect.handfist);
            }
            // Returns coordinates of fist
            var coords = detector.detect(video, 1);
            if (coords[0])
            {
                var coord = coords[0];
```

```
                /* Rescale coordinates from detector to video coordinate
space: */
                coord[0] *= video.videoWidth / detector.canvas.width;
                coord[1] *= video.videoHeight / detector.canvas.height;
                coord[2] *= video.videoWidth / detector.canvas.width;
                coord[3] *= video.videoHeight / detector.canvas.height;
            }
                else
                    fist_pos_old = null;
            }
        }
});
```

This library does not follow any specification, except the WebRTC one. It is, as opposed to voice recognition, harder to use and to understand. Indeed, it lacks an official documentation that specifies the proper way of using this library. There are only a few examples that ease the programmer's understanding but that do not really explain how it works. Moreover, it does not quite abstract the moderate complexity of the WebRTC API, which forces the programmer to consult external references about WebRTC and its usage. It also requires some more programming to detect anomalies but these will be discussed and overcome in chapter 7.

3.4.4. SLIDE CHANGE WITH REVEAL.JS

The reveal library (Willy-vvu, n.d.) is at first a high-level slideshow library developed by Hakim El Hattab (Hakimel, 2015). William W.Whu forked it on GitHub to extend it to movement detection. Although his library seems to be working correctly, it is limited to a slideshow usage with Hakim El defined interface. The code is open source, but not documented. This library could be useful to investigate in order to understand how camera data is processed and how the decision-making is implemented. However, there is not a clear structure, nor documentation. It is designed for slideshows predominantly and does not seem to be extendable to other uses. Therefore, it is quite unhandy to understand and exploit. Usage of this library would require a deep analysis of the code written by William. It is thus not of great interest in this thesis.

3.5. PERFORMANCE

The *objectdetect* library is useful but not powerful enough to be used in a real-life web application. The algorithm it uses has the benefit of not being too complicated and fast enough but the disadvantage of not being reliable. As for the example of the hand detection, it often detects a hand on the camera where there is nothing or an object not related to a hand. As a result, using the library needs some fault detection by the programmer to avoid a chaotic web application. These issues are discussed further in chapter 7.

Furthermore, gesture detection needs a lot of computing. On average, the camera captures 20 images per second, implying that the *objectdetect* algorithm is run 20

times per second. Detecting an object is not an easy task. The programmer needs to be careful with event handling. Processing too many events could drastically diminish the web application performances, freeze the page or worse, crash it. It is therefore recommended to decrease the event rate or to handle only a part of arriving events.

No rigorous tests were driven in this thesis as it is out of scope. However, if tests had to be driven to evaluate the library, they would have to take into account several aspects: environment, distance from the camera, camera configuration and processor availability. The environment factor is probably the most important. It includes light, human physiological parameters, background, surrounding movement and position with regard to the camera. The distance from the camera has a role to play as the object to be detected will be smaller or bigger and can influence the algorithm performance. The tests have to be equally conducted using a set of cameras with various hardware configurations as some may process more images per second or send bigger arrays of data. Finally, the processor availability describes its hardware configuration as well as the percentage dedicated to the web application. It is important to test these parameters both in a system using the library as it is and in a system adding an abstraction layer for fault detection (as presented in chapter 7).

4. INK WITH HTML5

4.1. INTRODUCTION

Ink modality is a communication channel that happens through a pen or stylus. This modality englobes handwriting, drawing, sketching… This modality can be quite handy on mobile devices where limited space for keyboard constrains a fast typing, whereas handwriting can be very fast.

There is unfortunately no current direct HTML5 solution for ink modality. Thus, we have to split the functionality into two parts: the drawing interface, which just accepts drawn input from the user and an ink recognizer capable of interpreting the meaning of the drawn input.

4.2. DRAWING INTERFACE

With HTML5 came the *canvas* element. A canvas element is nothing else but a graphics container. There is no behavior implemented by default. It requires thus some scripting. It is quite open to a lot of different uses, but always needs some JavaScript to perform a task. As the ink modality requires the ability to draw, a painting script (Gardnerp, 2012) is presented in appendix 4. This script is based on touch and is therefore compatible with mouse, hand touch and stylus.

The Canvas element has also the ability of being exported to an image, thanks to the “toDataURL” method. It returns an URL with a base64-encoded version of the canvas graphics. This image can then be sent to an analyzer that will be able to recognize handwriting, shapes or any drawing, depending on its complexity. The drawing interface only produces raw data and acts as an intermediary between the user and the image processor.

4.3. INK RECOGNITION

Once an image has been produced by the drawing interface presented in the previous section, it must be processed to extract an interpretation. The interpretation depends on what the user intended. A complete algorithm would be able to make the difference between ink inputs belonging to different domains. For instance, a smiley should be interpreted differently than an alphabet character or handwritten text. However, such algorithms are quite difficult to achieve and require an enormous amount of “models” to which the algorithm can compare the user data coming from the drawing interface. There is a specialized domain of the ink modality that is the handwriting recognition. This is also called OCR (Optical character recognition). There are lots of solutions and implementations of OCR engines but most of them are either embedded in mobile operating systems or proprietary: ABBYY FineReader (ABBYY, n.d.), Adobe Acrobat Professional (Adobe, n.d.), Nicomsoft OCR (Nicomsoft, n.d.)… There exists a small set of open source projects running OCR as

described here below. Some are run server-side (data is sent to a server which responds with text), some client-side (written in JavaScript).

4.3.1. TESSERACT

Tesseract (Smith, 1995) has been under development from 1985 to 1995 and is now owned by Google. It is an open source OCR engine written in C++ that began as a PhD research project. It has support for different languages and image types. There are some options to optimize recognition, such as:

- Treat image as single character
- Treat image as single word
- Treat image as block of text

According to (Smith, 2007), at the time Tesseract appeared, its results were seen as really promising. It was able to compete with the other commercial OCR engines. Although it is nowadays behind the leading other OCR engines when it comes to its accuracy, it has the advantage of offering a wide choice of features.

4.3.2. OCROPUS

This OCR package is written in Python and use recurrent neutral networks (Graves, et al., 2009) to achieve OCR (Tmbdev, 2015). This project is sponsored by Google (Breuel, 2007) and is based on Tesseract. It is however not designed for handwritten recognition but document analysis, such as converting a scanned text (which is an image) to actual text. The project has been forked to CLSTM which is written in C++ and shows better performance according to (Tmbdev, 2015) but is still mostly based on OCRopus.

4.3.3. CUNEIFORM

Cuneiform is an OCR engine developed in C++ (Pakkanen, 2011). It only accepts BMP files, which is not really problematic but could mean that a conversion algorithm is required. The project has not been updated since 2011 according to their official website. However, the source code can still be found on their website (Pakkanen, 2011). It is one of the rare OCR engines that are able to recognize text formatting such as boldness, italics and images (Ubuntu, n.d.).

4.3.4. GOCR

GOCR (Schulenburg, 2013) performs optical character recognition and works with PNM files, which are basically maps describing an image. Fortunately, we can we convert JPEG images to PNM files thanks to this Unix command:

```
djpeg -pnm -gray file.jpg > file.pnm
```

According to (Schulenburg, 2013), last version was released in 2013. It is once again designed for document analysis and recognition of scanned documents for example.

4.3.5. OCRAD.JS

Ocrad.js (Kwok, 2013) is an automatically generated JavaScript library using Emscripten (Kripken, 2015) and based on Tesseract, which makes it an OCR engine. It is quite simple of use (as illustrated in the code sample below) but quite limited, as it only consists of one function, according to documentation. The big benefit of this library is that it is directly embeddable in a web application. The text recognition is performed on the client's browser, which reduces network traffic and performance over slow networks.

```
var text = OCRAD(image);
```

4.3.6. WINDOWS INK ANALYSIS

Windows 8 has the ability to run desktop application with HTML5 and JavaScript. With this new feature came the *Windows.UI.Input.Inking* namespace, allowing optical character recognition. This is however not an open source library and is limited to windows 8 applications. This thesis being limited to HTML5 over web applications only, this library will not be discussed further.

4.3.7. $P

As opposed to all the libraries described to this point, $P is not an OCR engine. It is a 2D gesture recognizer (Vatavu, et al., n.d.). It can recognize any shape or drawing like houses, smileys or sketches as far as there are "models" that are defined for them. A "model" means in this context a pattern that is matched by a set of forms. For example, a smiley face's model would be two dots for the eyes and a curve for the mouth. This library only offers a set of 16 pre-defined models, which makes it unusable as it is. It is mandatory to add a wide set of models for recognition to work. On the other hand, the size of this set will directly impact the performance of the algorithm, caused by the necessity of matching the form to a wider set of models. Usage of this library would therefore require heavy work that would aim to define a complete set of models. Furthermore, the library performance would have to be tested to check if it does not collapse as the number of defined models grows.

4.4. PERFORMANCE

Tests were conducted for all previous OCR libraries. Three images were taken as inputs on the different systems listed in table 4.1 and described in the previous chapters. The first image is the easiest to recognize, as it is the closest to typed writing. The second is used to test handwriting and the third one to test language support (in this case, French). We can observe that the algorithms are globally not

really fit for handwritten recognition. Only the Cuneiform OCR engine has managed to correctly recognize the first image. The Ocrad.js algorithm, even if based on Tesseract, strangely does not produce similar results. Table 4.1 is only an example of the libraries outputs. Of course, the smallest variation can influence the algorithm and produce a completely different result. More accurate tests of this library should include a wider set of inputs and performing for each input multiple executions to observe if the libraries are deterministic.

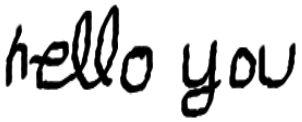
Algorithm			
Cuneiform	HELLO	held g.ou	BotdzouR
Tesseract	H ELLQ	(16090 gov	BONIOUR
GOCR	m ___o	__o gou	_oNJouß
Ocrad.js	WELLO	_eQQO _Du	_SONJOUk

Table 4.1. Performance of OCR engines

The $P algorithm has however shown real promising results as illustrated in table 4.2. The first four entries are pre-defined and already included in the library, while the two last ones were defined on the fly. The first column represents the input that is drawn by the user while the second column is the defined model that the $P algorithm has considered the closest to the drawn input. At first sight, the algorithm seems flawless and quite powerful. Nevertheless, it is necessary to perform tests to a bigger set of pre-defined models before it can be asserted as so. Indeed, the results of table 4.2 were conducted on a set of 18 models. It would be interesting to observe the results on a much wider set.

Drawn input	**Model Match**
	
	
	
	

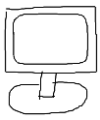
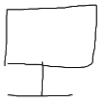
	
	

Table 4.2. $P performance evaluation

Considering the results of table 4.1, it is clear that the OCR engines that have been presented are not reliable enough to be used in a real web application. The OCR engine output would be too unpredictable. These engines will therefore not be addressed further in this thesis. As for the $P algorithm, even if it seems to be flawless and could be easily embedded in a multimodal library, it would require too much work (as described in section 4.3.7) to be carried out further in this thesis. It is however an interesting library to consider for shape recognition. It could be used for an OCR engine but as it only supports one letter at a time, it would require additional programming if the system needs to recognize multiple letters at a time.

5. THE EMMA MULTIMODAL LANGUAGE

5.1. INTRODUCTION

EMMA - Extensible Multimodal Annotation - is a XML format language created by the W3C as a standard for representation of data in a multimodal system (W3C, 2009). An EMMA document can contain up to three types of data:

- *Instance data:* usually generated by applications, this is data corresponding to the input of the multimodal application. For some types of modalities, it can happen to have several instances, like voice, where input can be ambiguous.
- *Data model:* model, structure of the document. This is usually pre-established to answer the needs of a specific application.
- *Metadata:* data passed as an annotation to the instance data, usually generated by the input processor.

EMMA can be used to describe speech, handwriting, pointing, keyboard input and combine all these inputs in a single document.

All examples in this section come from the W3C specification on EMMA.

5.2. GENERAL STRUCTURE

EMMA has a general structure that can be overviewed in the following example:

```
<emma:emma version="1.0"
        xmlns:emma="http://www.w3.org/2003/04/emma"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.w3.org/2003/04/emma
            http://www.w3.org/TR/2007/CR-emma-20071211/emma.xsd"
        xmlns="http://www.example.com/example">
      <emma:one-of id="r1" emma:start="1087995961542"
        emma:end="1087995963542"
        emma:medium="acoustic"
        emma:mode="voice">
            <emma:interpretation id="int1" emma:confidence="0.75"
                    emma:tokens="flights from boston to denver">
              <origin>Boston</origin>
              <destination>Denver</destination>
            </emma:interpretation>
            <emma:interpretation id="int2" emma:confidence="0.68"
                        emma:tokens="flights from austin to denver">
              <origin>Austin</origin>
              <destination>Denver</destination>
            </emma:interpretation>
      </emma:one-of>
</emma:emma>
```

All nodes start with the “emma:” keyword. Root node must be the <emma:emma>. Within this emma:emma node, one can add containers. W3C has defined three types of emma containers:

- <*emma:one-of*> elements in this container are mutually exclusive. In the above example, there would be only one alternative: either the flight is from Boston to Denver, either it is from Austin to Denver.
- <*emma:group*> elements in this container are complementary. For instance, changing color of an element requires pointing and voice.
- <*emma:sequence*> elements in this container should be considered sequentially. For instance, moving something across the screen requires firstly pointing to point an object, then pointing to where it should go.

Each container will then contain a list of *interpretations*. These nodes will contain the instance data (the actual data of the application), like utterances, coordinates or even a substructure of nodes that is specific to the application using it.

An EMMA document has then a simple abstract structure, summarized in Figure 5.1. This figure only represents a general overview of EMMA documents. For instance, containers can contain containers recursively and an EMMA root node can contain an interpretation (but only one). Sections 5.3 and 5.4 discuss further options and advanced elements.

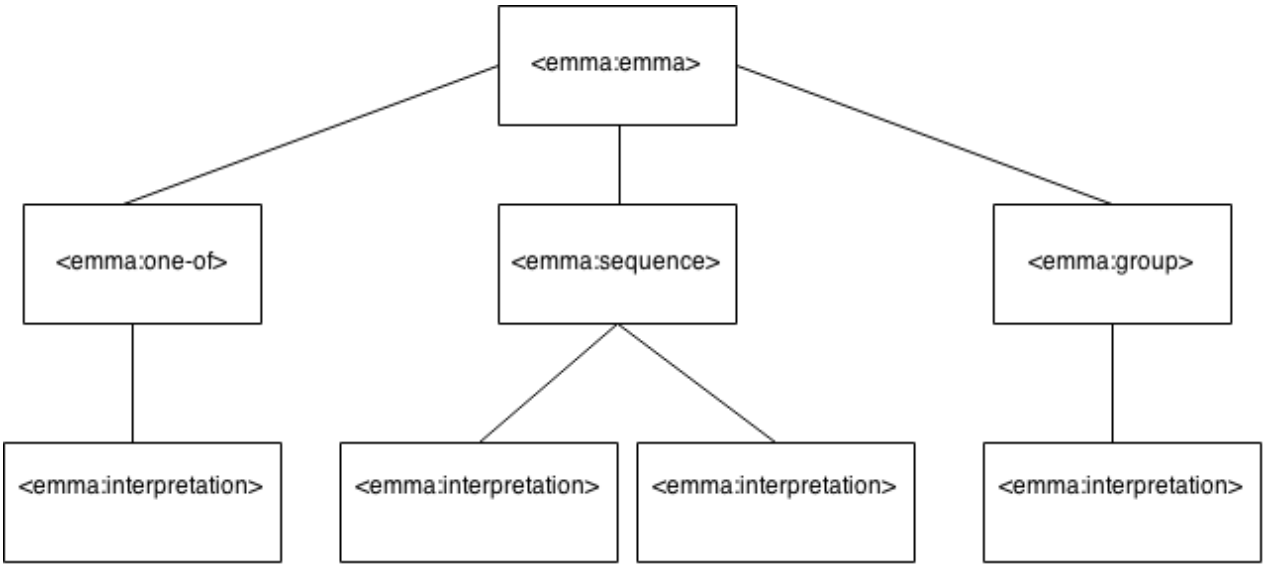


Figure 5.1. EMMA general structure

5.3. EMMA ELEMENTS

In this section will be discussed each EMMA element with principal options.

5.3.1. <EMMA :EMMA>

This element must always be the root element. An EMMA document not having <emma:emma> as root element will not be considered as a valid document. Direct children must be containers or a single interpretation. This node cannot be empty.

```
<emma:emma version="1.0" xmlns:emma="http://www.w3.org/2003/04/emma">
    ....
</emma:emma>
```

5.3.2. <EMMA:INTERPRETATION>

The data contained in this element is usually decided by the application type and is completely free. There can be multiple interpretations for one communication instance due to the fact that multimodal interactions frequently rely on non-deterministic modalities. An interpretation is basically an understanding of the world.

```
<emma:interpretation id="r1" emma:medium="acoustic" emma:mode="voice">
    ...
</emma:interpretation>
```

5.3.3. <EMMA :ONE-OF>

Emma one-of is the disjunctive element. It typically contains interpretation children but can also contain other containers. This is often used when data comes from a non-deterministic source or when it comes from different sources each returning a result. Elements of this container will be mutually exclusive.

This container can have an interesting attribute: *disjunction-type,* which can be set to four values:

- *recognition:* multiple results are naturally generated for ambiguous input. Homonyms in natural language are examples of this disjunction type.
- *understanding:* multiple results come from the input processor that is unsure of its results. For instance, a voice analyzer can hear “Boston” or “Austen” due to lack of accuracy.
- *multi-device:* results come from different devices.
- *multi-process:* results come from same device but by different interpreters.

```
<emma:emma version="1.0"
    xmlns:emma="http://www.w3.org/2003/04/emma"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2007/CR-emma-20071211/emma.xsd"
    xmlns="http://www.example.com/example">
```

```
  <emma:one-of disjunction-type="recognition"
      start="12457990" end="12457995"
      emma:medium="acoustic" emma:mode="voice">
     <emma:one-of disjunction-type="understanding"
         emma:tokens="boston">
       <emma:interpretation>
          <assert><city>boston</city></assert>
       </emma:interpretation>
       <emma:interpretation>
          <flight><dest><city>boston</city></dest></flight>
       </emma:interpretation>
     </emma:one-of>
     <emma:one-of disjunction-type="understanding"
         emma:tokens="austin">
       <emma:interpretation>
          <assert><city>austin</city></assert>
       </emma:interpretation>
       <emma:interpretation>
          <flight><dest><city>austin</city></dest></flight>
       </emma:interpretation>
     </emma:one-of>
  </emma:one-of>
</emma:emma>
```

5.3.4. <EMMA:GROUP>

Interpretations in an EMMA group container are complementary, meaning that they are all part of the same intent. The following example is given by the W3C specification, the classic “put-that-there” example combining voice and touch. The task is here to move an ambulance from one point to another by speaking “move that ambulance here”.

```
<emma:emma version="1.0"
    xmlns:emma="http://www.w3.org/2003/04/emma"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2007/CR-emma-20071211/emma.xsd"
    xmlns="http://www.example.com/example">
  <emma:group id="grp"
      emma:start="1087995961542"
      emma:end="1087995964542">
    <emma:interpretation id="int1"
      emma:medium="acoustic" emma:mode="voice">
      <action>move</action>
      <object>ambulance</object>
      <destination>here</destination>
    </emma:interpretation>

    <emma:interpretation id="int2"
      emma:medium="tactile" emma:mode="ink">
      <x>0.253</x>
      <y>0.124</y>
    </emma:interpretation>

    <emma:interpretation id="int3"
      emma:medium="tactile" emma:mode="ink">
      <x>0.866</x>
      <y>0.724</y>
```

```
    </emma:interpretation>
  </emma:group>
</emma:emma>
```

EMMA group element can also contain a *grouping-info* tag. This tag gives some information on how two interpretations of a same group are related. For example, we could consider the time that went by between two touches.

```
<emma:group-info>
    <ex:mode>temporal</ex:mode>
    <ex:duration>2s</ex:duration>
</emma:group-info>
```

5.3.5. <EMMA:SEQUENCE>

The EMMA sequence container constrains order of its children. In the “move that ambulance here” example in the previous section, the two touches could be included in a sequence container to remember the fact that one touch occurred before the other.

5.3.6. LATTICES

The <one:of> container represents a set of exclusive interpretations. Unfortunately, this is sometimes not powerful enough, mainly due to its potential combinatory explosion. For instance, if in a sentence where several words have different interpretations, the number of interpretation tags can grow drastically.

Lattices help to solve this problem. They can be seen as state machines where nodes are interpretations and arcs represent the links between them.

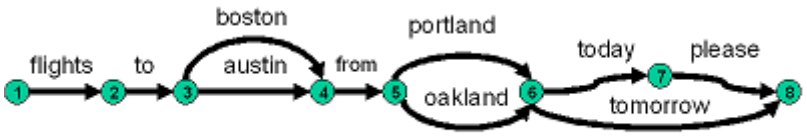


Figure 5.2. A lattice graphical representation (W3C, 2009)

The EMMA translation of Figure 5.2 could be (W3C, 2009):

```
<emma:emma version="1.0"
    xmlns:emma="http://www.w3.org/2003/04/emma"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2007/CR-emma-20071211/emma.xsd"
    xmlns="http://www.example.com/example">
  <emma:interpretation id="interp1"
    emma:medium="acoustic" emma:mode="voice">
    <emma:lattice initial="1" final="8">
      <emma:arc from="1" to="2">flights</emma:arc>

      <emma:arc from="2" to="3">to</emma:arc>
```

```
      <emma:arc from="3" to="4">boston</emma:arc>
      <emma:arc from="3" to="4">austin</emma:arc>
      <emma:arc from="4" to="5">from</emma:arc>

      <emma:arc from="5" to="6">portland</emma:arc>
      <emma:arc from="5" to="6">oakland</emma:arc>
      <emma:arc from="6" to="7">today</emma:arc>
      <emma:arc from="7" to="8">please</emma:arc>

      <emma:arc from="6" to="8">tomorrow</emma:arc>
    </emma:lattice>
  </emma:interpretation>
</emma:emma>
```

This shows how much the EMMA document can be shrunk when using lattices. Usage of one-of containers would require eight different interpretations.

The arc element can also carry metadata such as transition cost.

5.3.7. <EMMA:INFO>

This element is useful when some useful information needs to be remembered about the context of the EMMA document. The W3C specification gives the example of a conversation happening on the phone with a client asking for flights information:

```
<emma:emma version="1.0"
    xmlns:emma="http://www.w3.org/2003/04/emma"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2009/REC-emma-20090210/emma.xsd"
    xmlns="http://www.example.com/example">
  <emma:info>
    <caller_id>
      <phone_number>2121234567</phone_number>
      <state>NY</state>
    </caller_id>

    <customer_type>residential</customer_type>
    <service_name>acme_travel_service</service_name>
  </emma:info>

  <emma:one-of id="r1" emma:start="1087995961542"
      emma:end="1087995963542"
      emma:medium="acoustic" emma:mode="voice">
    <emma:interpretation id="int1" emma:confidence="0.75">
      <origin>Boston</origin>
      <destination>Denver</destination>
      <date>03112003</date>
    </emma:interpretation>

    <emma:interpretation id="int2" emma:confidence="0.68">
      <origin>Austin</origin>
      <destination>Denver</destination>
      <date>03112003</date>
    </emma:interpretation>
  </emma:one-of>
</emma:emma>
```

5.4. USEFUL ATTRIBUTES

5.4.1. EMMA:NO-INPUT

This attribute is used when no input has been detected. This is a boolean and is an attribute of an interpretation.

```
<emma:emma version="1.0"
    xmlns:emma="http://www.w3.org/2003/04/emma"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2009/REC-emma-20090210/emma.xsd"
    xmlns="http://www.example.com/example">
  <emma:interpretation id="int1" emma:no-input="true"

   emma:medium="acoustic" emma:mode="voice"/>
</emma:emma>
```

5.4.2. EMMA:LANG

The *emma:lang* interpretation attribute communicates the language of the input. This can be quite useful for voice and ink input. The attribute value can be anything but could benefit from following the ISO language code standard. Multiple languages can be given in the attribute value:

```
<emma:emma version="1.0"
    xmlns:emma="http://www.w3.org/2003/04/emma"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2009/REC-emma-20090210/emma.xsd"
    xmlns="http://www.example.com/example">
  <emma:interpretation id="int1"
      emma:tokens="arretez s'il vous plait"
      emma:lang="en fr"
      emma:medium="acoustic" emma:mode="voice">
    <command> CANCEL </command>
  </emma:interpretation>
</emma:emma>
```

5.4.3. EMMA:CONFIDENCE

This attribute can be heavily used for ambiguous inputs, particularly in one-of containers. This attribute indicates the quality of the input. Its value can vary between 0 and 1, 1 being maximum confidence and 0 minimal one.

```
<emma:emma version="1.0"
    xmlns:emma="http://www.w3.org/2003/04/emma"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2009/REC-emma-20090210/emma.xsd"
    xmlns="http://www.example.com/example">
  <emma:one-of id="nbest1"

    emma:medium="acoustic" emma:mode="voice">
```

```
    <emma:interpretation id="meaning1" emma:confidence="0.6">
      <location>Boston</location>
    </emma:interpretation>

    <emma:interpretation id="meaning2" emma:confidence="0.4">
      <location> Austin </location>
    </emma:interpretation>
  </emma:one-of>
</emma:emma>
```

This attribute can be both in an interpretation element and an element of the application namespace. For instance:

```
<emma:emma version="1.0"
    xmlns:emma="http://www.w3.org/2003/04/emma"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2009/REC-emma-20090210/emma.xsd"
    xmlns="http://www.example.com/example">
  <emma:interpretation id="meaning1" emma:confidence="0.6"
     emma:medium="acoustic" emma:mode="voice">
     <destination emma:confidence="0.8"> Boston</destination>
     <origin emma:confidence="0.6"> Austin </origin>
  </emma:interpretation>
</emma:emma>
```

5.4.4. DURATION

EMMA gives three optional attributes to record time of input.

- *emma:start* timestamp at which input started
- *emma:end* timestamp at which input stopped (must be after emma:start)
- *emma:duration* duration of input in milliseconds

This attribute can be useful for touch events when touching something for a long time can result into an action.

5.4.5. EMMA:MEDIUM AND EMMA:MODE

Emma:medium attribute tells which channel was used for input data. It is relative to the boundary that exists between the user and the machine. This attribute can be “acoustic”, “visual” or “tactile”.

Emma:mode attribute tells which particular mode was used to provide input. It gives the ability to differentiate different modes of communication that happen through the same medium. For instance, tactile medium contains the click, touch and ink modes. This is, as opposed to the medium attribute, let free by the W3C specification.

5.5. EARLY REVIEW

In this chapter has been discussed the EMMA language and its structure. Considering the fact that the language is a W3C standard, which is a predominant web-oriented standardization organization, it will be used as a communication format for the multimodal library developed in the next chapters and the corresponding usage examples. Usage of the EMMA standard is also a way of increasing compatibility with other existing multimodal libraries or future ones, thus reinforcing that choice.

A more advanced review will be presented in chapter 9. At this point, EMMA seems to be an adequate language for multimodality, which is the reason it will be discussed further in this thesis. However, it is a necessary process to review the language after it has been experienced through the development of the multimodal framework of chapter 7 and the usage examples of chapter 8.

6. LIBRARIES REVIEW

The previous chapters have discussed the current existing libraries built around HTML5 usable in a multimodal application context. Voice can be handled by using the webkitSpeechRecognition and speechSynthesis APIs, gesture with the WebRTC API coupled with the objectdetect JavaScript library and ink can be handled using various technologies detailed in table 6.1. The comprehensiveness of this list is however not guaranteed, as the web technologies are quickly expanding and are being updated.

The operability and performance of those technologies are discussed in details in the previous chapters and summarized in table 6.1. Although they are promising, they are generally not fit for business or professional web applications yet. They have to be improved or replaced by more performant technologies. Sadly, powerful systems exist but are generally not open to the public or owned and kept secret by companies. For instance, Google translation service has an ink recognizer that is not open and cannot therefore be used broadly.

Browser compatibility is another big concern too. All tests of this thesis have been conducted in Opera, Safari, Google Chrome, Mozilla Firefox and Internet Explorer, which are considered as the five current dominant existing browsers. After compatibility tests, it appears only Google Chrome supports all technologies. Therefore, the library developed in the next chapters is meant to be used in Google Chrome only.

Table 6.1 illustrates an overview of all libraries discussed in previous chapters and their corresponding characteristics. Browser compatibility is mentioned, as well as W3C standards consideration. The calculation type can happen on the server, such as the voice recognition API handled by Google servers, or on the client browser, such as gesture analysis. Ease of programming is also taken into account, as this can make a huge difference to the programmer. This last feature is evaluated both on understandability and customization. Finally, table 6.1 stipulates if source code is openly available or not.

MODALITY	TECHNOLOGY	COMPATIBILITY	CALCULATION TYPE	PERFORMANCE	PROGRAMMING	W3C	SOURCECODE	REMARKS
Speech Recognition	WebkitSpeechRecognition		Server	- Operational - Not fit for business	- Easy - Configurable	W3C Standard	~~**Open source**~~	EMMA generation not implemented
Speech synthesis	speechSynthesis		Opera: browser Chrome: Server	- Operational - Fit for business	- Easy - Slightly configurable	W3C Standard	~~**Open source**~~	Some features not implemented in Chrome
Handwriting recognition	Canvas + Ocrad.js		Browser	- Not operational	- Easy - Limited	/	**Open source**	
	Canvas + Cuneiform		Server	- Not operational	- Easy - Needs server side programming	/	**Open source**	Only BMP files
	Canvas + GOCR		Server	- Not operational	- Easy - Needs server side programming	/	**Open source**	Only PNM files
	Canvas + Tesseract		Server	- Not operational	- Easy - Needs server side programming	/	**Open source**	
	Canvas + $P		Client	- Operational	- Easy -Needs constructing a set of characters models	/	**Open source**	Limited set of pre-defined models
Touch Pointing	Touch/Pointing with JavaScript events		Browser	- Operational - Fit for business	- Easy - Configurable	W3C Standard	~~**Open source**~~	
Gesture	HTML5 media capture + objectdetect library		Browser	- Operational - Not fit for business	- Medium - Highly configurable	WebRTC W3C Standard	**Open source** Documented code but no documentation.	Low-level library. No movement analysis, just object detection
	HTML5 media capture + reveal.js (willy-vu)		Browser	- Operational - Fit for business	- Medium - Limited	WebRTC W3C Standard	**Open source** Undocumented code and no documentation.	High level library designed for slideshows

Table 6.1. Overview of existing technologies for multimodal web applications

Legend of table 6.1

- *Operational*: can be used directly and produces acceptable results.
- *Fit for business*: usable in a real-life application used by untrained end users.
- *Easy/Medium programming*: easy to use, easily understandable / Requires JavaScript skills and time for understanding.

Considering table 6.1 characteristics of different existing technologies, it is now possible to select a set of usable technologies that will be used in the next few chapters.

The voice modality includes both voice recognition and voice synthesis. Voice recognition exists only in Chrome and can be achieved by the *webkitSpeechRecognition* API. Despite the fact that it is not completely reliable and that it sadly does not generate EMMA, it is operational. It is easy to use and has got some interesting configuration options like language, maximum number of interpretations or results filtering. It requires an internet connection to function, which is not problematic considering the fact that browsers are generally meant to be connected to the Internet. Voice synthesis can be achieved by the *speechSynthesis* API available in both Chrome and Opera. It is instantiated by the browser, which section 2.4 has demonstrated to be a possible problem. Nevertheless, this library is easy to use and can be slightly configured in terms of language, pitch and rate. The Google implementation requires an Internet connection, which is again not problematic in web applications. Both APIs will therefore be used in the rest of this thesis.

The ink modality can be achieved in web browsers by combining the canvas element and an ink recognizer. Ink recognizers can be OCR (optical character recognition) systems such as Ocrad.js, Cuneiform, GOCR and Tesseract. Section 4.4 has demonstrated the poor results that were given by these engines and concluded that none of them were operational. They have easy interfaces and are all executed on a server except the Ocrad.js JavaScript library, which runs on the client's browser. The $P algorithm has shown some really promising results, whereas its weak point is its lack of pre-defined models (only 16). It can recognize any form (from drawing to alphabetic characters) but requires a set of models that it can compare the arriving input with. Section 4.4 has demonstrated that the work required to make this library functional is too substantial to be carried out further in this thesis. It is also important to notice that all these libraries are not compatible with standards like InkML (W3C, 2011). Considering the fact that there were no relevant libraries to be used in a multimodal framework, the ink modality will be ignored in the rest of this thesis.

The touch modality is probably the most performant and easiest modality to implement in browsers. It can be achieved by intercepting the *click* events of a web

page. This feature has been around for a long time now and have been implemented in a wide set of JavaScript frameworks. The JQuery will be used for this purpose in the rest of this thesis because it is easy of use. The JQuery framework is open source. However, the implementation of the JavaScript event listener is not always open source as this is the responsibility of the browser software, which is not guaranteed to be open source. This is why table 6.1 does not consider it open source.

Finally, the gesture modality can be achieved by coupling the WebRTC API and an additional layer of software that abstracts the complexity of image processing. Chapter 3 has illustrated that reveal.js was only designed for slideshows. Its open source factor would have been interesting if the source code was documented or commented so that it could be extended for other purposes. However, source code browsing has shown that this was not the case and has therefore not been carried out further in this thesis. The objectdetect library has on the contrary shown some interesting results. It is capable of recognizing a small set of body parts. It is compatible with Google Chrome, Opera and Mozilla Firefox. It is however limited to the object detection. There is for instance no capability of detecting movement through this library, which makes it on the other hand more configurable. The understanding of this library is not straight-forward like the voice synthesis and recognition APIs. It requires some additional layers of software to retrieve intelligence and add abstraction. This topic will be discussed further in chapter 8.

7. A MULTIMODAL FRAMEWORK

7.1. INTRODUCTION

The libraries presented in the previous chapters for building multimodal web applications can be encapsulated in a single web multimodal framework. This framework will have to offer a certain level of abstraction but give the opportunity for advanced configuration. Considering the fact that multimodality is always in expansion, the library should be extensible. It should be easy to add support for a new modality and to make it function with other multimodal systems such as fusion or fission engines.

The development of such a library has been a bottom-up process. In other words, it was implemented in parallel to the examples of chapter 8 that aim to illustrate this framework. The main reason for this choice is that multimodality is a complex domain, caused by the fact that technology for input modalities is not completely reliable. The modalities are inherently more complex, non-deterministic and still in research. Voice recognition for example is affected by the context in which it is running: surrounding noise, language, foreign accents... Current tools currently do not have the capacity of fixing all those problems, causing a natural unreliability. The second reason why input modalities from multimodal applications are not reliable is that the technology is still not powerful and efficient enough to handle worst-case scenarios. Gesture recognition is quite dependent on background and light for example. As a consequence, unreliability is at the center of every multimodal application, causing the necessity of taking it into account. The technology has to be tested and proven before it is embedded in a library. Unpredictable events can happen, as we will see through this chapter, making it necessary to have that bottom-up approach.

The multimodal framework that will be developed in this chapter will be based on a client and a server. The client will be the web browser, which will capture information and send it to the server. The server will have the task of processing all events and optionally respond with an action to perform. Figure 7.1 illustrates a global architecture of the future multimodal library. The client will be written in JavaScript, as the libraries presented in the previous chapters are all based on this language. The basic idea behind this library is that the client serves as an event listener and the server as an event processor. There is no real intelligence on the client. In fact, the client recorder, which acts both as a bridge between the client and the server and as a black box for modalities listening, is responsible for starting recognizers, listening to incoming events and sending those events to the server wrapped in an EMMA document. The APP1 and APP2 are scripts that are specific to a particular WEB application. The business logic of those scripts is completely free and is to be defined by the programmer with optional use of the server library. The server will receive all

events and infer actions from them. A server recorder written in PHP stores all events, forgets them after a defined delay and is able to detect combinations of different events. This last feature is also called the fusion engine. To ensure that events are remembered between two calls from the client recorder to the application scripts, the recorder's event list is stored into a session variable.

One of the possible consequences of this library is the danger of a slow internet connection or distant servers, resulting into delays in the response. However, it is established that the recording of voice and gesture already demands a considerable processor load. Even with optimization by parallelization and asynchronous programming, the web browser is still vulnerable to unresponsiveness, lag, overload and, in the worst-case scenario, crash. The main purpose of multimodality is to offer better human-machine interaction. Risking a crash of the web browser is not the way to go. Hence, web browsers run on a wide set of different hardware configurations, meaning that some machines will effectively have enough processing capacities but others will not. Thus, making all application logic part of the server is a way of lightening the web browser load. Furthermore, this architecture makes it easy to switch between fusion engines. Indeed, the client recorder sends EMMA documents to the server script. This script just has to understand and be able to parse EMMA documents to be directly usable with this library. Hence, the architecture enables to easily switch between languages. For instance, the server could be written in any web server language (Java, ASP, Ruby, Scala…). As long as they are able to parse the EMMA document sent by the client recorder, they can interact with the client.

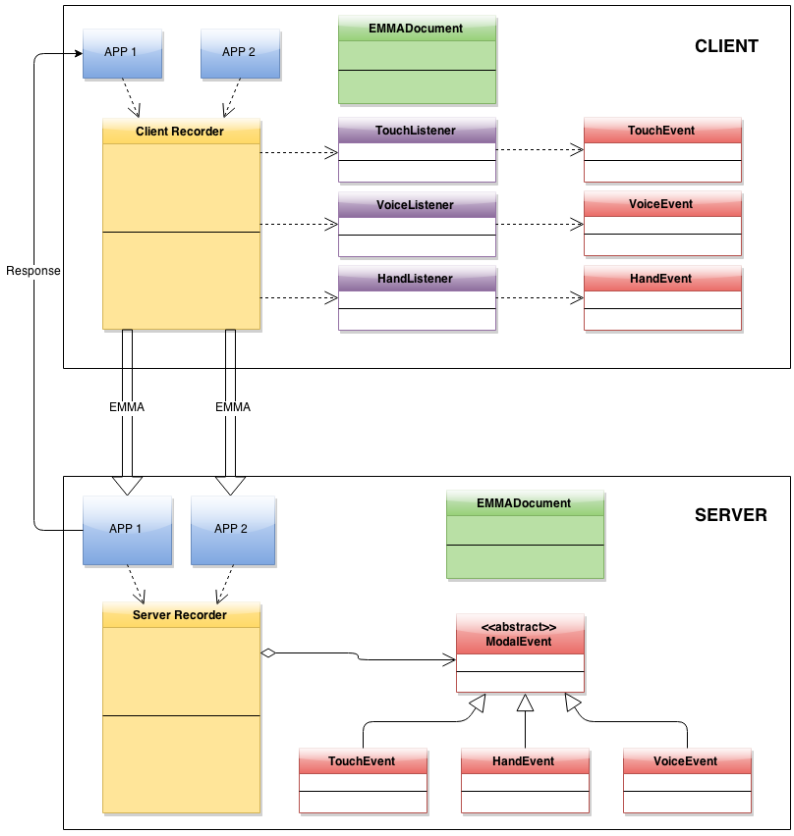


Figure 7.1. Multimodal framework skeleton

This architecture supports three modalities; voice, touch and hand gesture. It is however quite extensible to other modalities. For a new modality to be supported, the client recorder would have to implement two more methods: a method to start the modality, which would technically instantiate a new Listener class (represented in purple on Figure 7.1) and a method to intercept results of the listener. More details of implementation will be discussed in the next section. It is important to emphasize the fact that this library gives support for concurrent modalities. Multiple listeners can be launched at once and results retrieved and fused by the server script.

The next sections will concentrate on a concrete implementation of the library. Considering the fact that client and server parts of Figure 7.1 are quite different, they are discussed in different sections too. Section 7.4 will focus on the development of EMMA tools only.

7.2. JAVASCRIPT LIBRARY

The JavaScript library will be eventually run on the client's browser. The central piece of the client library, as illustrated on Figure 7.1 will be the client recorder. This entity should be highly configurable, as it will be the bridge between the actual modalities and the web application. JavaScript is not object-oriented but prototype-oriented. It is therefore not possible to directly translate them into classes. However, it is possible

to imitate object-oriented paradigms. For instance, a class can be defined as so (De la Marck & Pardanaud, 2012):

```
var Class = function(class_param_1, class_param_2)
{
    this.property1 = class_param_1 || 0;
    this.method1 = function(method_param_1, method_param_2)
    {

    }
}
var cl = new Class(5,10);
cl.method1(4,5);
```

This structure will be used in all future JavaScript classes. On the subject of the client recorder class, the constructor will take as a parameter an array of properties, which is quite standard in the JavaScript world. All properties can be omitted, thus it is important to create default values for each one of them. All possible parameters are listed in table 7.1. As for the features of this class, it will be able to handle three types of modalities: voice, touch and what is called in this thesis “hand”, which describes a hand location on the video camera and by extension the ability of controlling the web page by hand gesture. All available methods are described in Figure 7.2. The onXXXEvent methods are not to be called by the web application but rather by the listeners (illustrated in purple in Figure 7.1). These methods add some processing and algorithms to the raw recognizers to reduce as much as possible errors and hide as much as possible their complexity.

Name	Type	Default Value	Description
GLOBAL PARAMETERS			
onResult	Function(response: String)	Empty function	Callback user-defined function called when a result is returned by the server.
onEvent	Function(event) Event parameter can be any type of event integrated in the library	Empty function	Callback user-defined function called when an event occurs and before the data is sent to the server.
server_script	String	“”	Server script URL to be called by the recorder and to which data should be sent.
all_controls	Boolean	false	Displays visible controls on the bottom-right corner of the web page allowing control on all listeners. This property sets the voice_controls, voice_synthesis_controls, touch_controls and

			hand_controls properties to “true”.
VOICE MODALITY			
words	Array	[]	List of words, or group of words that will trigger the onEvent and onResult callback functions. In the case of an empty array, no filter is defined and the callback functions are called for each utterance.
min_words	Integer	null	If the “words” parameters is not defined, this property sets a minimal words to be spoken before callback functions are triggered. If null, no filter is defined. If the “words” parameter is defined, this property is ignored.
voice_controls	Boolean	false	Displays visible controls on the bottom-right corner of the web page that give the opportunity to start and stop voice recognition.
TOUCH MODALITY			
touch_selectors	Array	[]	Array of CSS selectors of HTML5 elements on the web page that make it relevant to trigger the callback functions. If left empty, there is no filter and all elements trigger the callback functions.
touch_controls	Boolean	false	Displays visible controls on the bottom-right corner of the web page that give the opportunity to start and stop touch recognition.
HAND MODALITY			
precision_interval	Array(2)	[15,60]	Couple of values describing an interval (in %). A new hand position will be considered relevant if distance to previous position is within this interval.
datasend_frequency	Integer	1000	Frequency in milliseconds at which data is sent to the server. In other words, this is the frequency at which the onResult function is triggered.
hand_frequency	Integer	50	Rate in milliseconds at which events are processed. In other

			words, if an event occurs less than *hand_frequency* ms after the previous event, it will be ignored.
hand_controls	Boolean	false	Displays visible controls on the bottom-right corner of the web page that give the opportunity to start and stop hand recognition.
display_video	Boolean	false	Displays video output on the top-right corner of the screen.
VOICE SYNTHESIS MODALITY			
ignoreSynthesis	Boolean	true	Mentions if voice synthesis should be ignored by the voice recognizer.
voice_synthesis_ controls	Boolean	false	Displays visible controls on the bottom-right corner of the web page that give the opportunity to stop, pause and resume voice synthesis.

Table 7.1. List of all available parameters for client recorder class.

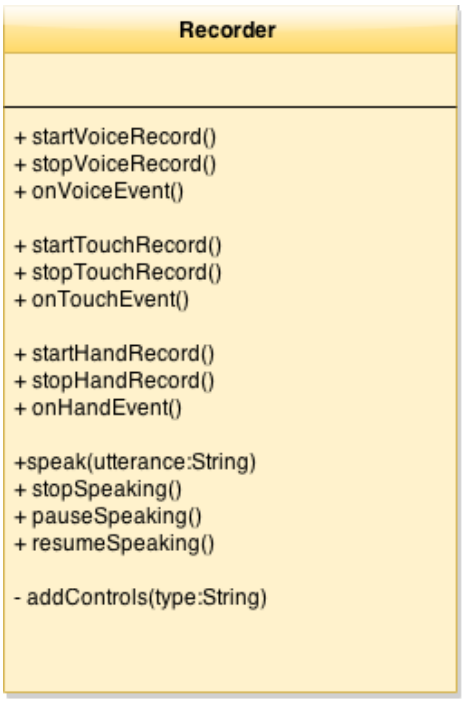


Figure 7.2. Available methods in the client recorder JavaScript class.

The different modalities are quite different, meaning that they process differently, at different rates, do not return the same information and have their own particularities

and problems. The library development was a continuous process and the problems were resolved as they came along thanks to the development of the examples discussed in chapter 8. Considering the fact that they act different, they have to be addressed differently. The next sections will discuss each modality separately.

7.2.1. VOICE RECOGNITION

The voice recognition has been implemented by using the Google *speechRecogniton* API presented in chapter 2. The client recorder class requires two more classes: a voice listener and a voice event class. Those are detailed in Figure 7.3.

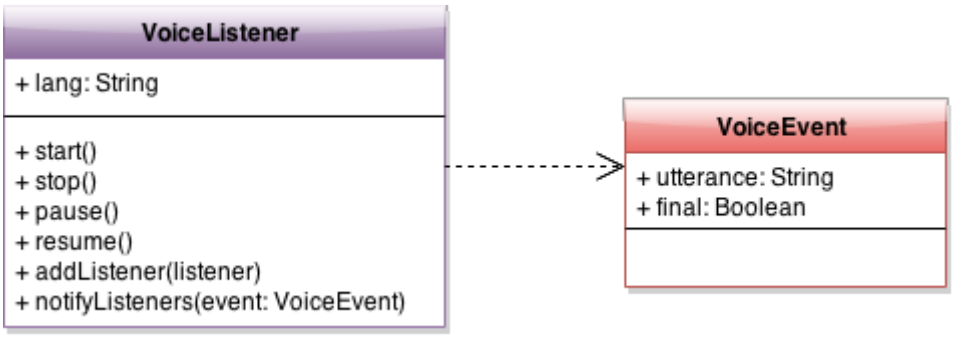


Figure 7.3. Voice listener and voice event classes.

The start method of the voice listener starts the *webkitSpeechRecognition* API and registers a callback function, which simply builds the utterance, encapsulates it into a *VoiceEvent* object and calls the *notifyListeners* method, which notifies all listeners of the incoming event. At this point, the client recorder class will be notified that something has been spoken through its *onVoiceEvent* method.

Although this appears to be quite straight-forward, it is not that simple. Actually, the *webkitSpeechRecognition* API is a continuous process, meaning that it sends back all interim results. For reminder, interim results are sent back while the user is still talking. They are most of the time not reliable and can change a lot depending on the other words of the sentence. Opposed to those interim results are final results, which always happen shortly after the user is done talking. Interim results are thus faster than final results. On the other hand, they are less reliable and can be repetitive. For instance, it often happens that speaking a sentence once generates multiple identical results. If not taken into account, the system could understand that the user spoke the sentence more than once and repeat the result routine faultily. Final results are slower but more reliable. We are therefore confronted to a dilemma: either we process all interim results, gain speed but lose reliability, either we process only final results, lose speed but gain reliability. The solution that was adopted in this thesis is a mix of them: all interim results are taken into account and trigger the *onEvent* method but only final results are sent to the server for processing. This does not quite

resolve the speed problem but, at least, the user gets some feedback on voice recognition.

Finally, the *onVoiceEvent* method will have to consider the values of the *words* and *min_words* parameters and decide whether or not the utterance should be ignored. Here below is the implementation of this *OnVoiceEvent* method.

```
this.onVoiceEvent = function(event)
{
    if(this.client_callback != null)
            this.client_callback(event);
    if(event.final === false)
        return;
    if((this.words == null && (this.min_words == null || this.min_words <=
event.utterance.split(" ").length))
        ||    (this.words != null &&
this.words.indexOf(event.utterance.trim()) != -1))
    {
        var callbackFunc = this.callback;
        var doc = new EMMADocument();
        var int1 = new Interpretation('acoustic', 'audio');
        int1.addData('<utterance>'+event.utterance+'</utterance>');
        doc.addInterpretation(int1);
        $.ajax({
                url: this.server_script,
                async: true,
                type: 'post',
                data: {
                    emma: doc.xmlString()
                },
                success: callbackFunc,
                error: function(xhr, text, error) {
                    console.log(error);
                }
        });
    }
}
```

7.2.2. VOICE SYNTHESIS

Voice synthesis can be manipulated through the *speak, stopSpeaking, pauseSpeaking* and *resumeSpeaking* methods of the client recorder class. Considering the fact that the code to launch a synthetized voice does not take much place, it is directly integrated in the client recorder class:

```
this.speak = function(utterance) {
    if(this.ignoreSynthesis)
        this.voice_listener.pause();
    var msg = new SpeechSynthesisUtterance(utterance);
    msg.lang = this.language;
    window.speechSynthesis.speak(msg);
    if(this.controls || this.voice_synthesis_controls)
        this.addControls("voice_synthesis");
    var thisobject = this;
```

```
    msg.onend = function(e) {
        if(thisobject.ignoreSynthesis)
            thisobject.voice_listener.resume();
    };
}
this.stopSpeaking = function() {
    window.speechSynthesis.cancel();
}
this.pauseSpeaking = function() {
   window.speechSynthesis.pause();
}
this.resumeSpeaking = function() {
   window.speechSynthesis.resume();
}
```

One problem that was observed during the development of the oral request example (section 8.4) is that the voice recognition API does take into account the synthetized voice. This can be a big issue in a web application that requires both modalities as it can cause an infinite loop. For instance, let's consider a web application listening to client requests and that responds by synthetized voice whenever a bad request has been made (for example, "your request is invalid"). If this last invalid request utterance is taken into account by the voice recognizer and considered as a bad request in turn, it will proceed again to that same synthetized utterance ("your request is invalid") and so on. Moreover, considering the fact that the voice synthesizer is instantiated by the browser itself and that all those bad request utterances are stored in a queue, the only way to clear this queue and stop the synthetized voice is to close the browser itself. The solution to this problem is to pause the voice listener while the synthetized voice speaks, then resume it when it stops. Resuming the voice recognizer is done after a second to ensure that they do not overlap.

7.2.3. HAND RECOGNITION

Hand recognition will be implemented with the *objectdetect* library presented in chapter 3. This has probably been the most difficult part of the library for two reasons. First, this library is less straight-forward to understand and requires some time to understand how to start the camera, register a listener and process the results. Secondly, the *objectdetect* library has not much documentation and can appear as quite buggy at first. It is highly unpredictable and the context in which it is executed influences a lot the results. Thus, it is important to add a layer on top of this library to resolve as much problems as possible before it is used in a real application.

The two classes that will be used for hand recognition are the *HandListener* and *HandEvent* classes illustrated in Figure 7.4. This structure is quite similar to the *VoiceListener* and *VoiceEvent* combo of Figure 7.3. This is done on purpose to keep a clean and common structure all through the multimodal library. The *HandEvent* class encapsulates two coordinates as percentages values. These coordinates are converted to percentages so that they can be extended to other purposes without carrying around the web camera size. For instance, if the position of an object that is moving on the camera needs to be detected for user feedback (like a cursor moving with the user hand), it is necessary to know the camera width and height so that proportionality can be respected. If the camera is wider than the area on which the feedback will be displayed (a section of the page for example), the coordinates will have to be reduced so that the coordinates stay in the feedback area. Therefore, to avoid using even more memory to include the web camera's dimensions, it is stored as a percentage. The hand width and height are stored as well for reasons that will be explained further. The *play* method of *HandListener* is a callback function triggered by the video support for HTML5 itself. The purpose of this callback function is to detect where the hand is and to send back the coordinates to the client recorder.

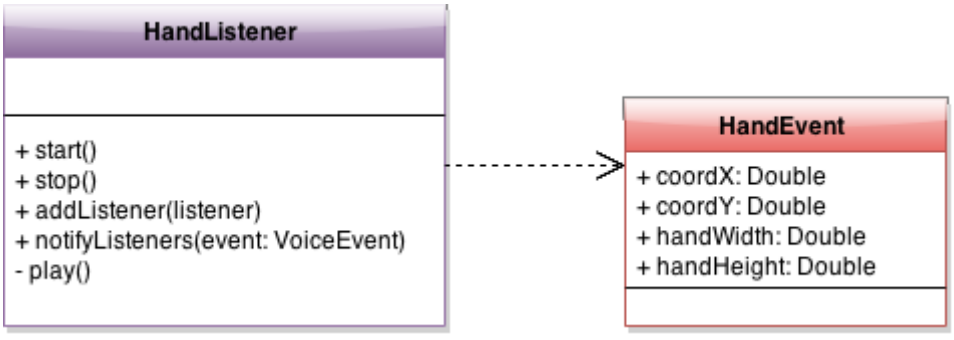


Figure 7.4. Hand listener and event classes

The coordinates are generated by the *detect* function of the *objectdetect* library. However, they need to be edited for them to make sense. The *detect* function

actually sends back an array of four values describing the position of a rectangle framing the actual hand:

- x coordinate of the top-left corner
- y coordinate of the top-left corner
- width of rectangle
- height of rectangle

The first action to perform will be to convert this information on a scale of one hundred (percentage). This is simply performed by applying the proportionality law.

After all this, it is possible to calculate the coordinates of the rectangle center, which is [x + width/2, y + height/2]. Finally, considering the fact that the camera reverses everything horizontally, meaning that when the user goes left the camera understands right, the x coordinate has to be reversed. As those coordinates have been converted in percentage, it is simple to perform an orthogonal symmetry: [100 – (x + width/2), y + height/2].

At this point, the hand listener *play* method will return the result to the *onHandEvent* method of the client recorder. The first action that this method will take is to check whether or not the event has to be taken into account according to the event rate parameter (see *hand_frequency* parameter of table 7.1). This helps to slow down a bit the processor load and avoid as much as possible crash or lag due to a processor struggling to keep up.

After this, the client recorder needs to judge if the coordinates that were returned are valid or not. Indeed, the *objectdetect* library is not bullet-proof. Depending on background and light, it can happen that a coordinate of a hand is returned, even if this hand does not actually exist. This is the purpose of the *precision interval* explained in table 7.1. The distance from the previous point at which the hand was located to the new point will be calculated using the Pythagoras theorem and compared with that *precision interval*. If the distance is outside the interval, it will not notify it to the user.

Another feature to take into account is to expand the coordinates. Indeed, the hand cannot be recognized in the whole camera area, as the hand has a certain size. If the hand is partially out of the camera area, it is not considered as a full hand. This problem is illustrated in Figure 7.5. Indeed, when the center of the hand is in the green zone, it will be recognized. When it is not in the green zone but still in the camera area, it will not be recognized because part of the hand is out of the camera area. Thus, the *objectdetect* library does not consider a wide part of the screen. When the hand is used to browse the whole screen, it is necessary to expand the coordinates that are returned.

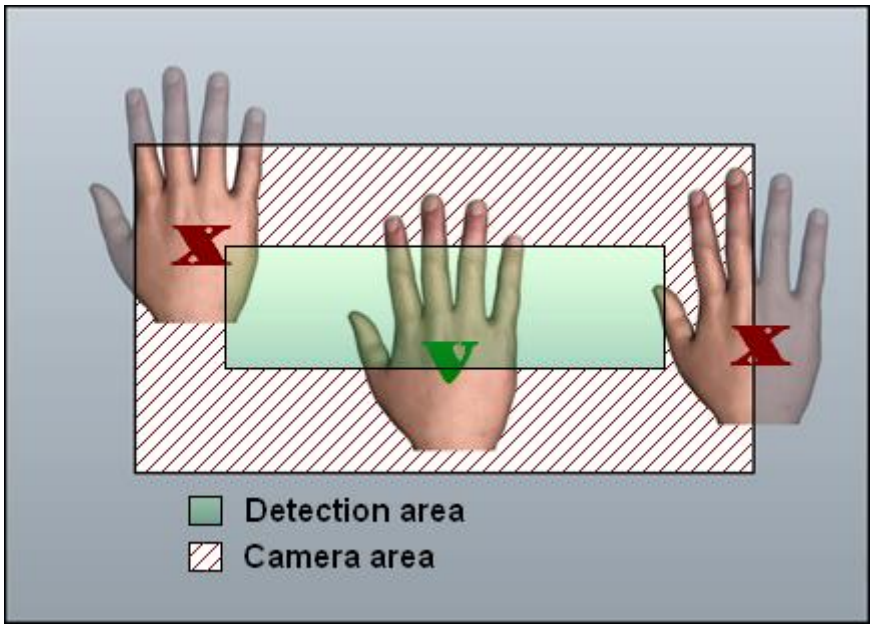


Figure 7.5. Hand detection area problem.

Hand source: (TurboSquid, n.d.)

This is where the hand width and height of the *HandEvent* object will be used. Indeed, it is mandatory to know the width and height of the hand to know how much the coordinates must be expanded. The hand width and height is expressed in %, making it compatible with the coordinates. Here below is the formula that is used to expand those coordinates. They are first shifted to the left and bottom so that the center of the screen ([50,50]) becomes the center of a Cartesian plane, then expanded to the whole plane and finally converted back in % coordinates.

```
event.coordX = (event.coordX-50)*(50/(50 - event.handWidth/2)) + 50;
event.coordY = (event.coordY-50)*(50/(50 - event.handHeight/2)) + 50;
```

Finally, the client recorder needs to send some data to the server. Send rate is defined by the *datasend_frequency* presented in table 7.1. Although it may appear at first sight that in case of an invalid coordinate, the data should not be sent to the server, it is not advised in practice. Indeed, the *precision interval* has a minimal value before it is considered as a valid move. For web applications that need to detect when the hand has stood still for a certain amount of time, there would be no way of triggering an event that could detect that. However, if data is sent to the server even when the coordinates are considered as invalid, the server can on its own group those coordinates, decide to ignore them or optionally take action and trigger an event. The example discussed in section 8.3 details how this could be implemented.

```
this.onHandEvent = function(event)
{
    // 0. Determine if event has to be treated or not
```

```
    if(this.hand_frequency > 0 && this.last_handevent > Date.now()-
this.hand_frequency)
        return;
    // 2. Check if coordinate is not absurd or has moved too little
    var goodCoordinate = false;

    if(this.old_position != null)
    {
        // Pythagore theorem for distance bewteen old point and new one
        var X = event.coordX;
        var Y = event.coordY;
        var oldX = this.old_position[0];
        var oldY = this.old_position[1];
        var diffX = Math.abs(X - oldX);
        var diffY = Math.abs(Y - oldY);
        var line_length = Math.sqrt((diffX*diffX) + (diffY*diffY));
        if(line_length > this.precision_interval[0] && line_length <
this.precision_interval[1])
            goodCoordinate = true;
    }
    else
    {
        goodCoordinate = true;
    }
    // Expansion
    event.coordX = (event.coordX-50)*(50/(50 - event.windowWidth/2)) + 50;
    event.coordY = (event.coordY-50)*(50/(50 - event.windowHeight/2)) + 50;
    // 3. If good coordinate, call visual callback
    if(goodCoordinate)
    {
        if(this.client_callback != null)
            this.client_callback(event);
        this.old_position = [event.coordX, event.coordY];
        this.last_handevent = Date.now();
    }
    // 4. Send data to server
    if(this.datasend_frequency > 0 && Date.now()-this.datasend_frequency >
this.last_datasent
        && event.coordX >= 0 && event.coordX <= 100
        && event.coordY >= 0 && event.coordY <= 100)
    {
        // Sending data to server
        var callbackFunc = this.callback;
        var doc = new EMMADocument();
        var int1 = new Interpretation('visual', 'hand');
        int1.addData('<X>'+event.coordX+'</X>');
        int1.addData('<Y>'+event.coordY+'</Y>');
        doc.addInterpretation(int1);
        $.ajax({
                url: this.server_script,
                async: true,
                type: 'post',
                data: {
                    emma: doc.xmlString()
                },
                success: callbackFunc,
                error: function(xhr, text, error) {
                    console.log(error);
```

```
                }
            });
            this.last_datasent = Date.now();
        }
}
```

7.3. SERVER LIBRARY

The global architecture makes the server script completely free of implementation. The user of the JavaScript library could just implement a server script parsing the EMMA document that is sent and take action accordingly. However, as the goal of this chapter is to implement a working framework to ease the development of multimodal web applications, a server library written in PHP will be presented here below.

Whatever the language used, a multimodal framework that should work with the JavaScript library introduced in previous section should be expected to fulfill the following goals:

1. It should provide with a mechanism to store all incoming events easily.
2. It should give support for EMMA parsing and building.
3. It should take into account time and give support to detect and remove expired events.
4. It should provide an easy mechanism to match an utterance to an EMMA lattice (speech state machine).
5. It should optionally offer a fusion engine algorithm.
6. It should optionally provide with event filtering mechanisms.
7. It should optionally be able to generate multimodal application server scripts through configuration files or a user-friendly interface.
8. It should optionally be able to save events on a persistent storage device for statistics purposes.
9. It should optionally offer a fission engine algorithm.
10. It should optionally offer a support to keep track of the user's context.
11. It should optionally offer a dialog manager algorithm.

The PHP server library illustrated in Figure 7.6 offers a working framework that supports point 1, 2, 3, 4 and 5. The events are stored in the *events* property of the server recorder. They can be filtered through the *getEvents, getLastEvent, removeOfType* methods. The *reset* method deletes all stored events. Expired events are deleted through the *cleanup* method (which is called by most of the other methods) depending on the retention period. Actually, each event stored in the server recorder has a time property that mentions at what time it was added in the recorder. This property and retention period are coupled to detect when an event has become too old.

Finally, the server recorder has a *detectCombination* method that implements the fusion engine. It takes two parameters. The first one is an array of *ModalEvent*, giving some criteria on the events to match. For instance, giving a *VoiceEvent* with utterance property set to “test” will retrieve the last “test” voice event. There are special constants in *TouchEvent, HandEvent* and *VoiceEvent* called ANY_XXX that make it possible to match on any event, like any coordinate or any utterance. The second parameter is a function. If defined, it is executed as an additional test once a match is found.

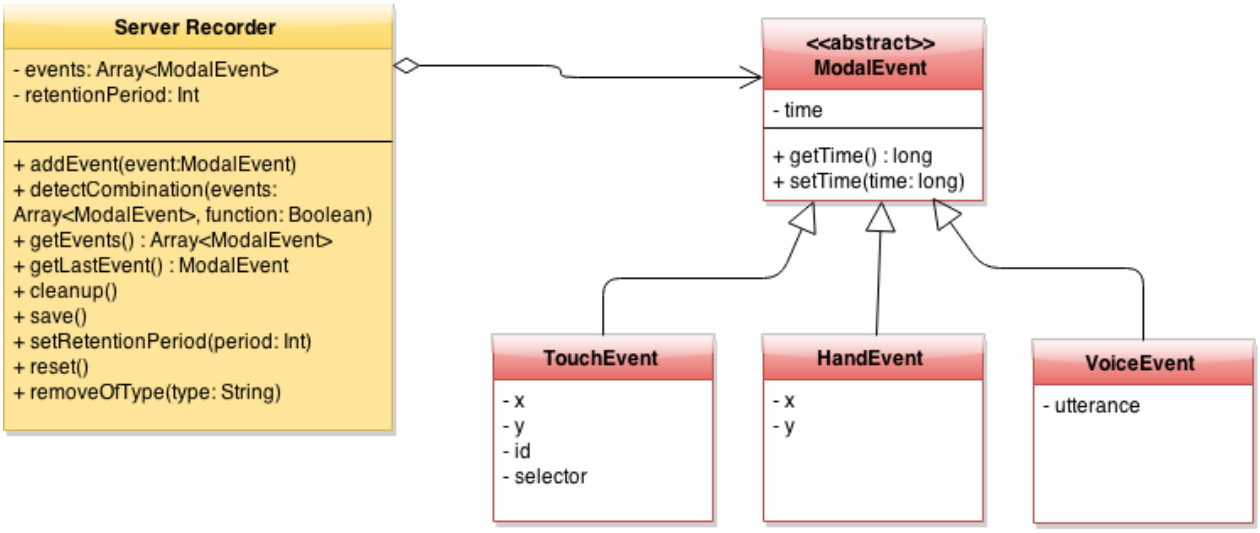


Figure 7.6. Server library structure.

Figure 7.6 is a model that can be used in multiple languages, as long as they support the object-oriented paradigm. One element has been omitted in this library: the EMMA support. Actually, this has been omitted too in the JavaScript library. It will be discussed in the next section.

7.4. EMMA LIBRARY

Chapter 5 has presented EMMA as a suitable language for multimodal web applications. Therefore, it is an important feature of a multimodal framework to offer EMMA generation and parsing tools. These tools have to be available in both the JavaScript and server libraries. Indeed, the JavaScript library will have to generate those documents and the server will have to parse them.

Figure 7.7 illustrates the structure of the EMMA library. It is built on the composite pattern. The EMMADocument is the englobing structure, containing a list of nodes. To avoid an illegal EMMA structure where nodes would be added without being embedded in an interpretation or a container, it only has got the methods to add a container or an interpretation. The library acts on strings to implement the children, so that exporting it for network travel is not too expensive. As described in the EMMA

specification, there are three types of containers: one-of, group and sequence. Each of them can contain other containers recursively or interpretations.

The Lattice class describes an EMMA lattice. It can be imported and built from a string. This string can be built by reading it from a file or by using the EMMANode class to dynamically add arcs. This class transforms a XML string into an in-memory state machine. The "*accepts"* method can then analyze if a group of words (typically a sentence) is valid considering that state machine. The *areLinked* method returns true if two nodes are connected by an arc. The *equals* method of the VoiceEvent class is able to match an utterance to a lattice by using the Lattice class and calling its *accepts* method.

Through this library, goals 2 and 4 of goals list for a successful server library from section 7.3 are achieved. Indeed, it enables EMMA parsing and generation. It also provides the framework with a mechanism to match an utterance to an EMMA lattice. Although the PHP library respects the structure given in Figure 7.7, the JavaScript library slightly differs. This is due to the fact that JavaScript is prototype-oriented and not object-oriented. All classes are there but there are no explicit relationships between them, such as heritage.

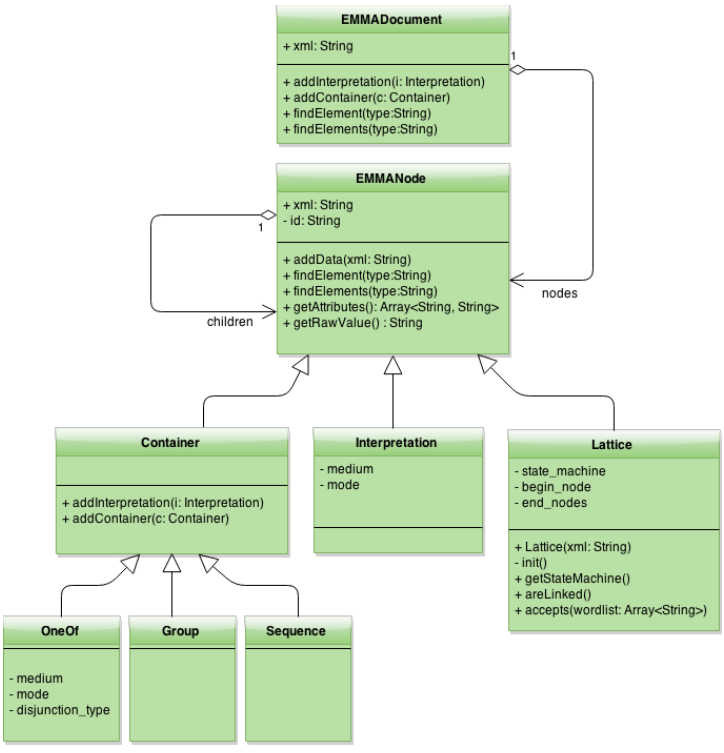


Figure 7.7 EMMA library structure

8. EXAMPLES

8.1. INTRODUCTION

The goal of this chapter is to illustrate how the framework introduced in the previous chapter can be used. These examples were built in parallel to the framework of chapter 7. The goal was to gradually implement the framework by elaborating these examples. The reader will observe a progressive complexity among the given examples. They were all tested in Google Chrome over an Apache server capable of interpreting PHP scripts. Each example is introduced by presenting its goal, some issues that could be experienced and finally how they will be resolved and implemented. These examples only give an overview, the full source code can be found in the appendix.

Four examples will be presented in this chapter. The first one is only an introduction to the framework, which aims to implement a simple “put that there” by using touch and voice. The second example is meant to explore as a single modality the ability to control a cursor with the hand and eventually select objects on the screen. The third example introduces the capability of understanding the user through the voice modality in a natural language by defining an EMMA lattice, or a state machine. The last example is a combination of all previous examples combining gesture and voice modalities and that will ultimately be able to perform the same tasks as Richard Bolt’s “put that there” experiment (Bolt, 1979).

8.2. PUT-THAT-THERE WITH VOICE AND TOUCH

8.2.1. PROBLEM DESCRIPTION

Multimodality is often illustrated using the put-that-there example. It requires two modalities: touch and voice. The user will be speaking “put that there” while touching on the screen an object and the emplacement where to put this designated object. This example seems easy at first sight but still needs some thinking on some problems that may be encountered. The first question to be asked would be the order in which we receive those events. If we represent the touch to designate the object by a target and the touch to designate the destination by an arrow, we could have all configurations shown on Figure 8.1 (and many more).

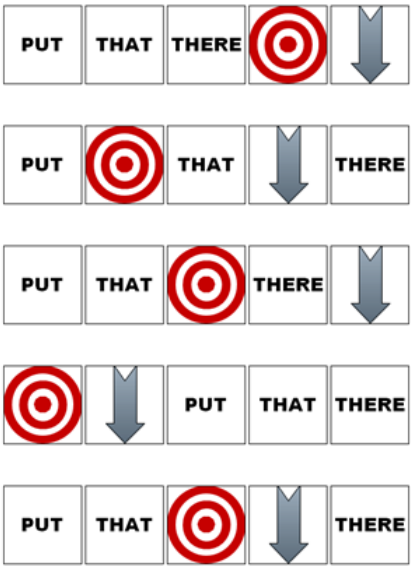


Figure 8.1. List of possible put-that-there event arrivals

The arrival of the modality events can depend on both the user and the speed of the event listener, as well as hardware. For example, touch is much faster than voice recording and analysis. Thus the “put that there” could be uttered while touching the screen but received after all touch events due to slow voice recognition.

The backend application will therefore have to manage all possible configurations to be able to process the event in any case. Due to important number of different configurations, it will be easier to follow a rule that will always be true, based on common sense. The user will have to pronounce the “put that there” sentence. All words of this sentence will be treated as a group as they would be in a real-life context. Moreover, it is certain that the user will always touch the object to be moved *before* showing where to put it. The destination will thus always follow the object designation.

Thanks to these hypotheses, we can reduce the set to three configurations illustrated in Figure 8.2.

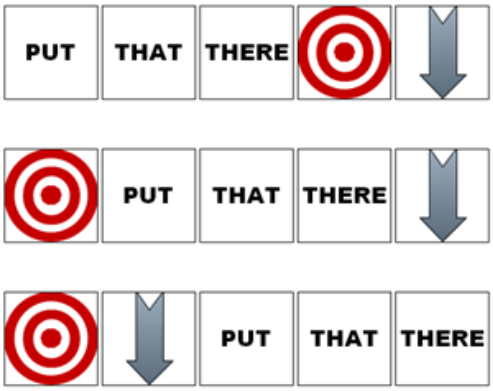


Figure 8.2. Simplified set of put-that-there event arrivals

The second question to be asked is “what may move and where?”. The main purpose of asking this question is to be able to distinguish the first touch and the second touch *for sure*. For instance, if the user makes the first touch to designate an object, speaks the “put that there” command and then stops for any reason, the web application will have recorded a partial match to execute the multimodal command. All it needs is another touch. If now the user starts a new put-that-there command (unaware the web application has not forgotten the previous one) and proceeds to a new touch, this last one will be considered as a “destination” touch (not a designation one) and cause the object to move mistakenly. This issue is illustrated in Figure 8.3 where the red arrow is mistakenly mapping a “first touch” to a “second touch”.

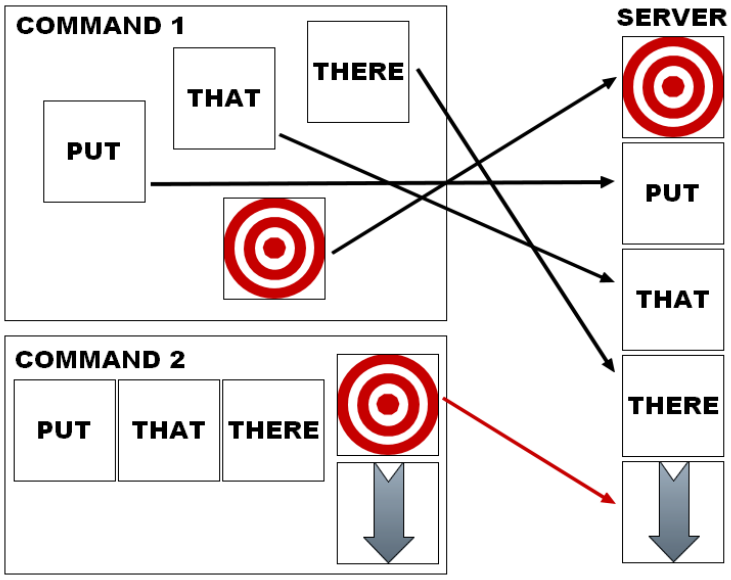


Figure 8.3. Mistaken merge of two independent commands

Giving a lifespan on the server events is a way to reduce this problem. However, the problem could occur again if the user is too quick to start the second command. This

problem can be completely avoided by defining two disjoint sets: the moveable objects and the recipients in which the moveable objects can be moved.

The problem illustrated at Figure 8.3 can be generalized to this question: “to what command do this event belong?”. In the library developed in chapter 7, the server acts as a recorder of all interesting events, which can rapidly become messy as events are recorded. Without smart management, events could be mistakenly grouped together to match a multimodal command despite the fact that they were not in the same command in real life. For instance, if command 1 was not completed and command 2 was completed, command 1 should not use an event of command 2 to complete itself. It should just be abandoned. This is resolved by setting a bigger priority to more recent events. This way, old commands do not merge with new ones.

8.2.2. IMPLEMENTATION

This example will be implemented using the library developed in the previous chapter. There will be two scripts launched for this purpose. One will be written in JavaScript and will execute on the client’s side, the other one will be written in PHP and will execute on the server. The JavaScript script will be responsible for launching the recording of voice and touch and take action when the server sends a command to move an object. The PHP server script will be responsible for remembering all events and detect valid “put-that-there” combinations.

The client script sends an EMMA document to the server as described in the previous chapter using Ajax. The PHP script will have to parse this EMMA document to retrieve important information. If a combination is found, it will send back a command to the client script as a JSON array with information about the object to move and where to move it.

The client script will look like this:

```
var record = new Recorder({
    server_script:   'put_that_there.php',
    onResult: function(result){
        var response = JSON.parse(result);
        if(!(Object.keys(response).length === 0))
        {
            if(response.action == 'MOVE')
            {
                $('#'+response.from.ID).css('left', response.to.X+'px');
                $('#'+response.from.ID).css('top', response.to.Y+'px');
            }
        }
    },
    words: ['put that there'],
```

```
    touch_selectors: ['.container', '.moveable']
});
record.startVoiceRecord();
record.startTouchRecord();
```

The server script is more complex. The first thing to do is to instantiate a recorder that is stored in session between two Ajax calls.

```
$recorder = new Recorder();
$recorder->setRetentionPeriod(10);
```

The retention period is the time an event is kept in memory before it is considered as obsolete. The retention period is set to ten seconds.

The next thing to do will be to add a new arriving event to this recorder. As we received an EMMA document, it will need some parsing, which will retrieve the event type (touch or voice) and instantiate the event depending on its type:

```
$doc = new EMMADocument($_REQUEST['emma']);
$data = $doc->findElement('emma:interpretation');
if($data->getAttributes()['emma:medium'] == 'acoustic')
{
    $event = new VoiceEvent($data->findElement('utterance')-
>getRawValue());
    $recorder->addEvent($event);
}
if($data->getAttributes()['emma:medium'] == 'tactile')
{
    $X = $data->findElement('X')->getRawValue();
    $Y = $data->findElement('Y')->getRawValue();
    $id = $data->findElement('ID')->getRawValue();
    $selector = $data->findElement('selector')->getRawValue();
    $event = new TouchEvent($X, $Y, $id, $selector);
    $recorder->addEvent($event);
}
```

Finally, the server script will need to detect a combination of a “put that there” utterance and two touches. The first touch will have to be on a moveable object and the second one on a container (to resolve issues discussed in the previous section). The *detectCombination* method of the recorder will be used to answer those needs. It will return a combination of three events: an event with the “put-that-there” utterance and two touch events. These two touch events will have to be in a particular order: first the touch on the moveable object, then the touch on the container. This way, the problem of faulty merge of two independent commands illustrated in Figure 8.3 will be fully avoided.

```
$match1 = new VoiceEvent("put that there");
$match2 = new TouchEvent(TouchEvent::ANY_COORDINATE,
TouchEvent::ANY_COORDINATE, TouchEvent::ANY_ELEMENT, '.moveable');
$match3 = new TouchEvent(TouchEvent::ANY_COORDINATE,
TouchEvent::ANY_COORDINATE, TouchEvent::ANY_ELEMENT, '.container');
```

```
$combo = $recorder->detectCombination(array($match1, $match2, $match3),
function($match){
    return ($match[1]->getSelector() == '.moveable' && $match[2]-
>getSelector() == '.container');
});
if($combo != NULL)
{
    // Found a match!
    $result = array(
        'action'    =>  'MOVE',
        'from'      =>  array(
                            'X'     =>  $combo[1]->getX(),
                            'Y'     =>  $combo[1]->getY(),
                            'ID'    =>  $combo[1]->getID()
        ),
        'to'        =>  array(
                            'X'     =>  $combo[2]->getX(),
                            'Y'     =>  $combo[2]->getY(),
                            'ID'    =>  $combo[2]->getID()
        )
    );
    echo json_encode($result);
}
else
{
    echo json_encode(array());
}
```

The full example code can be found in the appendix. The code here above needs the code of the developed library of previous chapter to run and only runs on Google Chrome. It was tested on a laptop using a mouse click instead of touch.

It is also important to consider that this example uses the *webkitSpeechRecognition* API of Google and can be therefore quite slow. It has some important limits directly related to the speech recognizer itself such as slow processing time, inoperability in noisy areas… It is therefore not directly suitable for websites with unaware users.

8.3. OBJECT SELECTION WITH GESTURE

8.3.1. PROBLEM DESCRIPTION

The purpose of this example is to give the ability to the user of selecting an object on the web page using only gesture. The movements of the hand will move a cursor on the screen. The object will be selected if the hand stands still for three seconds. Once again, the problem seems easy to solve. However, a few issues must be taken into account to create a reliable application.

Firstly, the objectdetect library presented in the chapter on gesture is not completely reliable. It sometimes detects a hand where there is none, causing wrong cursor feedback on the screen. The cursor will jump across the screen faultily. For this purpose, we define an interval of values. Each new arriving set of coordinates is compared to the previous one. If distance between these two sets of coordinates is not contained in the defined interval, the cursor will not move. The interval has been set to [15,60] after a battery of tests which proved this interval as a suitable one.

Another topic to take into account is the event rate. The camera sends a lot of images per second and thus can cause processing issues and overhead if all events are taken into account. To remedy to this problem, events will be ignored while other will be taken into account following a timing policy. Arbitrarily, events will be taken into account every 50ms. Moreover, data will be sent to a server script to detect a hand standing still. To reduce overhead of network, there will also be an event rate based on time (every second) at which data will be sent to the server script.

8.3.2. IMPLEMENTATION

This example will be implemented by using a server-side and client-side script. The client-side script (written in JavaScript) will be responsible for starting the hand listener and give visual feedback to the user. A secondary cursor will be presented on the screen, moving as the user moves his hand. When the hand stands still, which will be determined by the server script (written in PHP), the user will have a feedback on how much time left the hand has to stand still before the intended object will be selected.

The following code shows in the *onEvent* callback function how the visual feedback is handled. The multimodal library sends coordinates, which are used to move the cursor position on the screen.

```
var record = new Recorder({
        server_script:   'hand_select.php',
        controls: true,

        onEvent: function(event){
```

```
            if(event.coordX < 0)
            event.coordX = 0;
            if(event.coordY < 0)
                event.coordY = 0;
            if(event.coordX > 95)
                event.coordX = 95;
            if(event.coordY > 90)
                event.coordY = 90;
            $('#hand_cursor').css('top', event.coordY+'%');
            $('#hand_cursor').css('left', event.coordX+'%');
            $('#X').text(event.coordX);
            $('#Y').text(event.coordY);
        }
    });
record.startHandRecord();
```

Events are sent to the server every second (default value of multimodal library). Whenever an event is received by the server script, it is added to the recorder. Events are received in an EMMA document:

```
<?php
    // 1. Initialising recorder
    $recorder = new Recorder();

    // 2. Adding new events
    $doc = new EMMADocument($_REQUEST['emma']);
    $data = $doc->findElement('emma:interpretation');
    if($data->getAttributes()['emma:medium'] == 'visual')
    {
        $X = $data->findElement('X')->getRawValue();
        $Y = $data->findElement('Y')->getRawValue();
        $newevent = new HandEvent($X, $Y);
        $recorder->addEvent($newevent);
    }
?>
```

The next step is to detect if there are events in the recorder that can be grouped together as describing a hand standing still. For this purpose, we collect all the events that were gathered and get all those that are similar to the new arriving event. Events are similar when the *equals* method of the *HandEvent* class returns true.

```
<?php
    // 3. Detecting matches
    $events = $recorder->getEvents();
    $similar = array();
    foreach(array_reverse($events) as $ev)
    {
        if($ev->equals($newevent))
            $similar[] = $ev;
        else
            break;
    }
?>
```

The *similar* array contains a list of all events whose coordinates are close enough to the new arriving event ones. This array will always at least contain one element: the new one. If this is the only event of the list, we send back a *HAND_STILL* command mentioning a 0 second STILL event. When there are more events, we send back a HAND_STILL command mentioning the hand was still for a certain amount of time, which is determined by calculating the time difference between the first element of the *similar* array and the last one.

```
<?php
    if(count($similar) >= 2)
    {
        // There was already at least an event just before the one we added
        $result = array(
            'action'    =>  'HAND_STILL',
            'hand'      =>  array(
                                'X'     =>  $newevent->getX(),
                                'Y'     =>  $newevent->getY(),
                                'NB'    =>  count($similar),
                                'time'  =>  $newevent->getTime()
                                            - end($similar)->getTime()
- 1
            )
        );
    }
    else
    {
        // There is only the one we just added
        $result = array(
            'action'    =>  'HAND_STILL',
            'hand'      =>  array(
                                'X'     =>  $newevent->getX(),
                                'Y'     =>  $newevent->getY(),
                                'NB'    =>  0,
                                'time'  =>  0
            )
        );
    }
?>
```

The response is then encoded to a JSON array and sent back to the client script that will perform a “countdown” action in case the hand was still for more than one second. It will display three circles. Empty ones describe the remaining seconds before the designated object will be selected. This is illustrated on Figure 8.5.

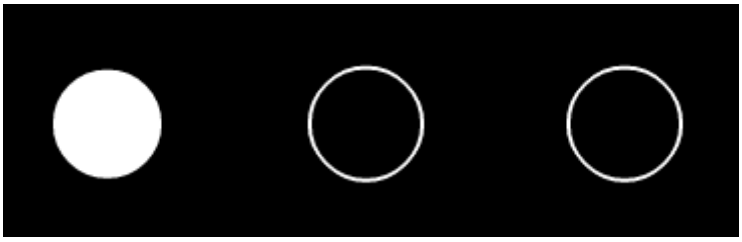


Figure 8.5. Two remaining seconds visual feedback to the user

Obviously, this visual countdown will have to disappear whenever the user moves the hand. When the third circle is filled, the script will virtually select the designated object by using the jQuery nearest library (Gilmoreorless, 2015).

```
onResult: function(result){
    var response = JSON.parse(result);
    if(!(Object.keys(response).length === 0))
    {
        if(response.action == 'HAND_STILL')
        {
            $('#still').text(response.hand.time);
            if(response.hand.time == 0)
            {
                $('#progress').hide();
                $('#one').removeClass('full').addClass('empty');
                $('#two').removeClass('full').addClass('empty');
                $('#three').removeClass('full').addClass('empty');
            }
            else
            {
                $('#progress').show();
                if(response.hand.time >= 1)
                    $('#one').removeClass('empty').addClass('full');
                if(response.hand.time >= 2)
                    $('#two').removeClass('empty').addClass('full');
                if(response.hand.time >= 3)
                {
                    $('#three').removeClass('empty').addClass('full');
                    var square = $.nearest({
                        x: response.hand.X*$(window).width()/100,
                        y: response.hand.Y*$(window).height()/100
                    }, '.moveable');
                    square.css('opacity', '1');
                }
            }
        }
    }
}
```

8.4. ORAL REQUEST

8.4.1. PROBLEM DESCRIPTION

This example aims to implement a web application hearing a request from the user and responding to this request appropriately. The EMMA specification example about flights (W3C, 2009) will be the base of this example. The user will speak a sentence asking for flights between two airports. This sentence will have to follow a certain structure as shown in Figure 8.6. Four airports are taken into account: Portland, Boston, Austin and Oakland. Those have been chose according to the EMMA specification about lattices.

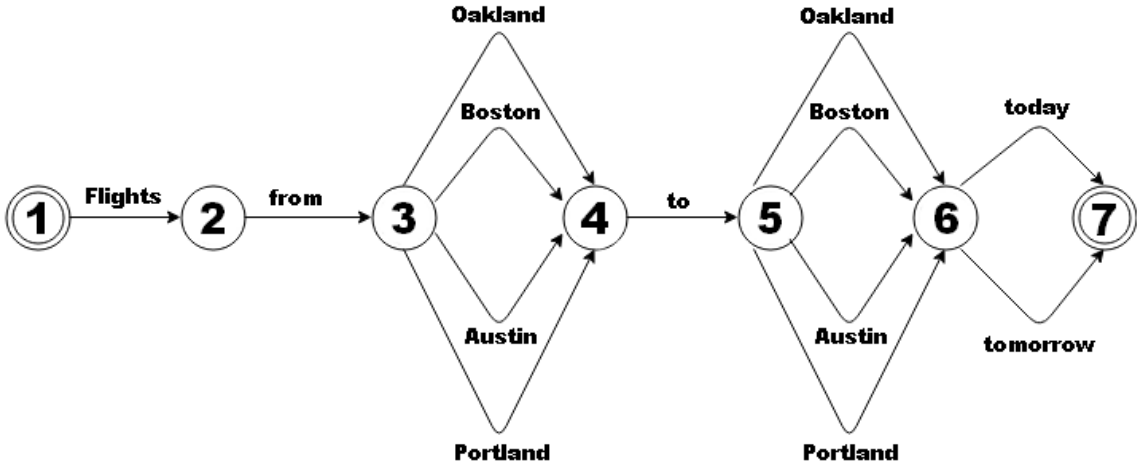


Figure 8.6. Oral request for flights structure

The speech structure represented in Figure 8.6 will be used as a model in the oral request web application to detect a valid sentence and to list all available flights according to the given request. A sentence has to start form point 1 and stop at point 7. Here below is an exhaustive list of all sentences that can be spoken by the user and will be considered as valid:

1. Flights from Oakland to Boston today
2. Flights from Oakland to Boston tomorrow
3. Flights from Oakland to Austin today
4. Flights from Oakland to Austin tomorrow
5. Flights from Oakland to Portland today
6. Flights from Oakland to Portland tomorrow
7. Flights from Boston to Oakland today
8. Flights from Boston to Oakland tomorrow
9. Flights from Boston to Austin today

10. Flights from Boston to Austin tomorrow
11. Flights from Boston to Portland today
12. Flights from Boston to Portland tomorrow
13. Flights from Austin to Oakland today
14. Flights from Austin to Oakland tomorrow
15. Flights from Austin to Boston today
16. Flights from Austin to Boston tomorrow
17. Flights from Austin to Portland today
18. Flights from Austin to Portland tomorrow
19. Flights from Portland to Oakland today
20. Flights from Portland to Oakland tomorrow
21. Flights from Portland to Boston today
22. Flights from Portland to Boston tomorrow
23. Flights from Portland to Austin today
24. Flights from Portland to Austin tomorrow

This speech structure can be directly translated into an EMMA lattice (as explained in chapter 5). These lattices are quite helpful as they drastically reduce the size of the model. The list here above illustrates how tedious it is to define the model, with only four airports. As a result, the following EMMA model can be defined:

```
<emma:emma
version="1.0"xmlns:emma="http://www.w3.org/2003/04/emma"xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2007/CR-emma-20071211/emma.xsd"
    xmlns="http://www.example.com/example">
  <emma:interpretation id="interp1"
    emma:medium="acoustic" emma:mode="voice">
    <emma:lattice initial="1" final="8">
      <emma:arc from="1" to="2">flights</emma:arc>

      <emma:arc from="2" to="3">from</emma:arc>

      <emma:arc from="3" to="4">boston</emma:arc>
      <emma:arc from="3" to="4">austin</emma:arc>
      <emma:arc from="3" to="4">portland</emma:arc>
      <emma:arc from="3" to="4">oakland</emma:arc>

      <emma:arc from="4" to="5">to</emma:arc>

      <emma:arc from="5" to="6">portland</emma:arc>
      <emma:arc from="5" to="6">oakland</emma:arc>
      <emma:arc from="5" to="6">boston</emma:arc>
      <emma:arc from="5" to="6">austin</emma:arc>

      <emma:arc from="6" to="7">today</emma:arc>
      <emma:arc from="6" to="7">tomorrow</emma:arc>
    </emma:lattice>
  </emma:interpretation>
</emma:emma>
```

8.4.2. IMPLEMENTATION

The implementation of this example will be done by using the multimodal library developed in chapter 7. A JavaScript script will be written to launch the voice recording and recognition while a server script will receive voice utterances wrapped in an EMMA document and try to match them to the lattice model defined in the problem description.

The client script resembles the one given in the put-that-there example, except that the limitation to reduce network traffic will be on a count of words in the utterance instead of the utterance itself. The client script could alternatively match the utterance to the lattice model but this would make the server useless and everything would be performed on the client's side. For an example with four airports, this is not too problematic. Nevertheless, an example with thousands of airport would require more resources, which include the download of the EMMA lattice model, its parsing and the in-memory state machine required to effectively match the utterance. Giving this responsibility to the server makes the web application lighter and avoids these problems.

Whenever the client script receives a response from the server, it will display them in the result table to be consulted by the user. If there are no results or the two given airports are equivalent, it will be notified to the user through a synthetized voice. If the request was incorrect, no action will be taken.

```
$(document).ready(function(){
    var record = new Recorder({
        server_script:   'oral_request.php',
        min_words: 6,
        controls: true,
        onResult: function(result){
            if(result.trim() == "EMPTY")
            {
                $('.line').hide();
                record.speak("No flights are available for your request");
                $('#flights').find("tbody").html('');
                $('#flights').find("tbody").append("<tr><td colspan='4'>No
results

found</td></tr>");
            }
            else if(result.trim() == "INVALID")
            {
                $('.line').hide();
            }
            else if(result.trim() == "SAME")
            {
                $('.line').hide();
                record.speak("Please provide us with two different
airports");
                $('#flights').find("tbody").html('');
```

```
                $('#flights').find("tbody").append("<tr><td colspan='4'>No
results

found</td></tr>");
            }
            else
            {
                $('.line').hide();
                var response = JSON.parse(result);
                if(!(Object.keys(response).length === 0))
                {
                    for (var i = 0; i < response.length; i++)
                    {
                        var from = response[i].from;
                        var to = response[i].to;

                        var date = response[i].date;
                        var hour = response[i].hour;
                        $('#flights').find("tbody").html('');

$('#flights').find("tbody").append("<tr><td>"+from+"</td><td>"+to+"</td><td
>"+date+"</td><td>"+hour+"</td></tr>");
                    }
                }
            }
        },
        onEvent: function(event){
          $('#utterance').text(event.utterance+"("+Date.now()+")");
        }
    });
    record.startVoiceRecord();
});
```

As for the server script, it will receive the EMMA document containing the voice utterance, storing it into a container if the given event is an acoustic one and then try to match this utterance with the EMMA lattice model defined in the problem description. This matching actually happens in the *equals* method of the *VoiceEvent* class using the *accepts* method of the EMMADocument class. This method is fully described in chapter 7.

```
<?php

    // 1. Initialising recorder
    $recorder = new Recorder();
    $recorder->setRetentionPeriod(10);

    // 2. Adding new events
    $doc = new EMMADocument($_REQUEST['emma']);
    $data = $doc->findElement('emma:interpretation');
    if($data->getAttributes()['emma:medium'] == 'acoustic')
    {
        $event = new VoiceEvent($data->findElement('utterance')-
>getRawValue());
        $recorder->addEvent($event);
    }
```

```
    // 3. Detecting matches
    $model = new
EMMADocument(file_get_contents("doc/flight_lattices.xml"));
    $lattice = $model->findElement("emma:lattice");

    $combo = $recorder->detectCombination(array($lattice));

?>
```

If there is a match, the server script will extract from a database (in this example, just an in-memory hard-coded array of flights) a JSON-encoded list of flights answering the user request. If the destination airport is the same as the departing one, it will respond with a “SAME” message to communicate to the client script that departure and destination airports were equivalent. If no results were found, an “EMPTY” message will be sent back. The recorder is reset after each event to reduce its size because remembering events is useless in this web application.

```
<?php
// Found a match, command is legal
    $result = array();
    $words = explode(" ", $event->getUtterance());
    $from = $words[2];
    $to = $words[4];
    $day = $words[5];
    if($from == $to)
    {
        // No need of browsing if two same airports
        echo "SAME";
    }
    else
    {
        foreach(getAllFlights() as $flight)
        {
            if($day == 'today')
                $date = '07-07-15';
            else
                $date = '08-07-15';
            if($from == $flight['from'] && $to == $flight['to'] && $date ==

$flight['date'])
            {
                $result[] = $flight;
            }
        }
        if(empty($result))
            echo "EMPTY";
        else
            echo json_encode($result);
        $recorder->reset();
    }
?>
```

8.5. THE “PUT-THAT-THERE” REVISITED

8.5.1. PROBLEM DESCRIPTION

The purpose of this example is to implement a system similar to Richard Bolt’s “put that there” experiment (Bolt, 1979) combining voice and gesture for the ability to draw and move shapes across a screen. Furthermore, this example has the benefit of showing a full example of multimodality and how modalities can be combined together using the framework presented in chapter 7.

Figure 8.7. Bolt’s “Put that there” experiment (Bolt, 1979)

This experiment was performed in 1979, followed by a paper published by Richard A. Bolt (Bolt, 1980). He designed an office with embedded cameras, microphones, a joystick and a wall-sized camera. The purpose was to create a “real-space” environment that the end user could manipulate naturally without typing on a keyboard in front of a terminal. A microphone recorded what the user said and was able to recognize short sentences with a pre-defined vocabulary of maximum 120 words. The user was able to point at a particular point on the screen thanks to the ROPAMS system (Remote Object Position Attitude Measurement System) built on magnetic fields. The system was able to create shapes of different sizes and different colors, move, copy, delete and edit their properties by combining voice and pointing. Users could select shapes by pointing at them or by describing them (for instance: “move that left of the red square”).

The implementation here below will basically imitate the features and interactions that were used in Bolt's experiment. The user will be able to draw three types of shapes (circles, squares and triangles) in three different colors (blue, red, green). Shapes can be optionally small or large, depending on the user's request. Once a shape is created, the user will be able to move it to a different place by pointing the shape or by describing it (for instance, "move the large blue square there"). The implementation will lack one feature from Bolt's experiment, which is the "move below another shape" feature.

This example requires two input modalities. The first one is voice, where the user utters its command. The second one is gesture, which will be implemented in this example using a hand recognizer as in the object selection example of this thesis (section 8.3). The HTML5 canvas element will be used for effective shape drawing on the web page.

For this example to work, it is also going to need a speech model as in previous example so that the engine is able to interpret user input as a command. As described above, the user will need to be able to create shapes and to move them. Creation can be performed by uttering sentences like:

- "Draw a red square here"
- "Create a large blue circle there"
- "Draw small red square here"
- "Draw a small green circle there"
- "Create a red triangle"
- ...

The previous list is obviously a subset of all possible examples. To be able to define a full set of possible sentences, EMMA lattices can be used. We can therefore define a model for shape creation as being a sentence beginning with "create" or "draw", followed by an optional "a" and "large" or "small" size description. Following this, the user will have to speak a color (limited to blue, red and green in this example), a shape (square, triangle, circle) and a "here" or "there" keyword to end the sentence.

Shape shifting can be achieved by speaking sentences like:

- "Move that here"
- "Put that there"
- "Move that red circle here"
- "Put this large green triangle there"
- "Put the small blue square here"

Once again, this is a non-exhaustive list of possible user commands. A full speech model for shape shifting would have to start with "put" or "move", followed by a

designation proposition such as “that”, this” or “the”, optional additional information such as the size, color and shape and finally a “there” or “here” word to close the sentence.

The full utterance model is illustrated in Figure 8.8. This model will be used for the EMMA lattice in the effective implementation.

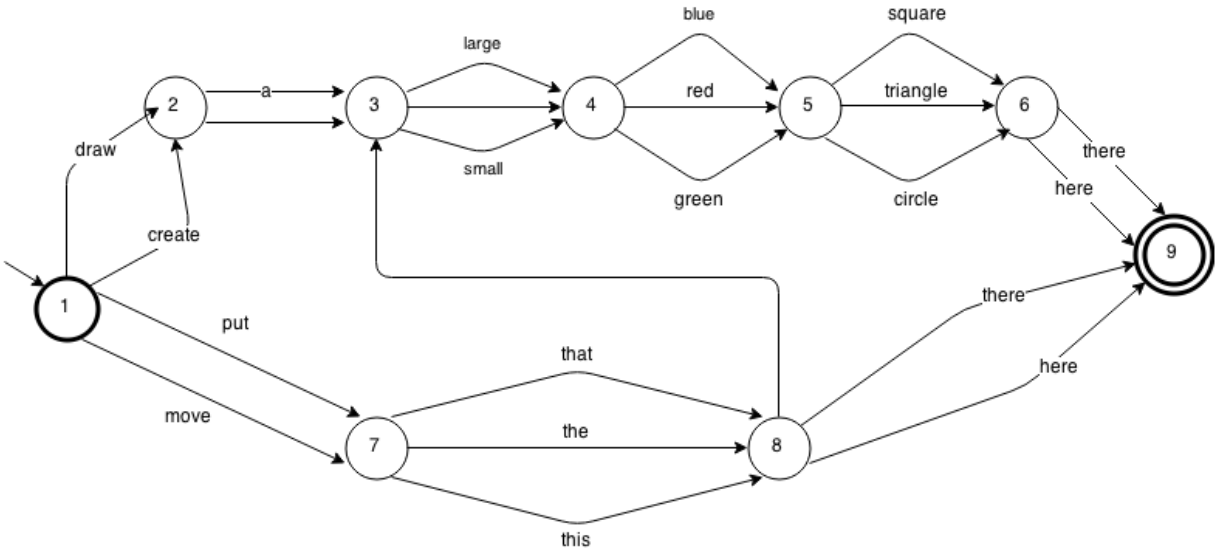


Figure 8.8. Speech model for shape drawing.

Figure 8.8 can be directly translated to this EMM document:

```
<emma:emma
version="1.0"xmlns:emma="http://www.w3.org/2003/04/emma"xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/04/emma
     http://www.w3.org/TR/2007/CR-emma-20071211/emma.xsd"
    xmlns="http://www.example.com/example">
  <emma:interpretation id="interp1"
    emma:medium="acoustic" emma:mode="voice">
    <emma:lattice initial="1" final="9">
      <emma:arc from="1" to="2">draw</emma:arc>
      <emma:arc from="1" to="2">create</emma:arc>

      <emma:arc from="1" to="7">move</emma:arc>
      <emma:arc from="1" to="7">put</emma:arc>

      <emma:arc from="2" to="3">a</emma:arc>
      <emma:arc from="2" to="3"></emma:arc>

      <emma:arc from="3" to="4">large</emma:arc>
      <emma:arc from="3" to="4">small</emma:arc>
      <emma:arc from="3" to="4"></emma:arc>

      <emma:arc from="4" to="5">blue</emma:arc>
      <emma:arc from="4" to="5">red</emma:arc>
      <emma:arc from="4" to="5">green</emma:arc>
```

```
      <emma:arc from="5" to="6">square</emma:arc>
      <emma:arc from="5" to="6">triangle</emma:arc>
      <emma:arc from="5" to="6">circle</emma:arc>

      <emma:arc from="6" to="9">here</emma:arc>
      <emma:arc from="6" to="9">there</emma:arc>

      <emma:arc from="7" to="8">that</emma:arc>
      <emma:arc from="7" to="8">this</emma:arc>
      <emma:arc from="7" to="8">the</emma:arc>

      <emma:arc from="8" to="3"></emma:arc>

      <emma:arc from="8" to="9">here</emma:arc>
      <emma:arc from="8" to="9">there</emma:arc>

    </emma:lattice>
  </emma:interpretation>
</emma:emma>
```

8.5.2. IMPLEMENTATION

The implementation of this example is actually a mix of all previous examples. All problems encountered in previous examples and the solutions to fix those will not be described here again. However, this example comes with a few other problems and sometimes needs adjustments regarding other examples due to the fact that they are fused. For instance, we could rely in the “put that there” example on a simple detection of a combination of two touch events and an utterance, while the combination needed in these examples varies, depending on whether the user creates a shape or wants to move it.

Once again, the example is programmed using the library developed in chapter 7 and uses a server and client-side script. The client-side script will be responsible for starting voice and hand gesture recording, user visual feedback and effective drawing and manipulation of the shapes. The server script however will be responsible for matching a spoken sentence to the speech model described in the previous section and translating the sentence into a JSON encoded command that can be understood by the client-side script.

The client-side script needs to instantiate the recorder and set up the canvas environment. The canvas environment will be the container for shape drawing. Its width, height and background color will be defined at first:

```
$(document).ready(function(){
var canvas = document.getElementById('canvas');
var canvasWidth = 1200;
var canvasHeight = 300;
var canvasBackground = "black";
var marginX = ($(window).width()-canvasWidth)/2;
```

```
var marginY = ($(window).height()-canvasHeight)/2-25;
var last_coords = [0,0];
var last_selected = null;
var shapes = [];


    $('#canvas').attr('width', canvasWidth);
    $('#canvas').attr('height', canvasHeight);
    $('#canvas').css('background-color', canvasBackground);

    var record = new Recorder({
        server_script:   'color_shapes.php',
        controls: true,
        min_words: 2,
        datasend_frequency: 3000,
        onResult: function(result){

        },
        onEvent: function(event){

        }
    });
    record.startHandRecord();
    record.startVoiceRecord();

});
```

To have a visual feedback about the hand movement, which is quite important to the user, the *onEvent* function will get the current offset as a percentage, convert it into pixels by calculation based on the window dimensions and finally move the cursor adequately to these calculations. As the *onEvent* function is called on every event, including voice and gesture, a test must be done to know which kind of event needs to be handled:

```
onEvent: function(event){
            if(event instanceof HandEvent)
            {
                if(event.coordX < 0)
                event.coordX = 0;
                if(event.coordY < 0)
                    event.coordY = 0;
                if(event.coordX > 95)
                    event.coordX = 95;
                if(event.coordY > 90)
                    event.coordY = 90;
                var X = event.coordX/100*$(window).width();
                var Y = event.coordY/100*$(window).height();
                $('#hand_cursor').css('left', (X-20)+'px');
                $('#hand_cursor').css('top', (Y-20)+'px');
                $('#X').text(event.coordX);
                $('#Y').text(event.coordY);
                last_coords = [event.coordX, event.coordY];
            }
            else
            {
                $('#utterance').text(event.utterance+"("+Date.now()+")");
```

```
            }
        }
```

To ensure maximal precision and reduce problems due to latency, the last hand position is saved client-side (in the *last_coords* variable).

At this point, the script is recording voice and gesture and sending all the data to the server. The server script then needs to handle all these incoming events and return the appropriate action. The first job is to start the event recorder and add the new arriving event. The arriving event type is determined thanks to the EMMA medium attribute. If the incoming event is a hand event, the script detects if it has been standing still for a few seconds, and sends back a “STILL” command when this the case:

```
// 0. Dependencies
require_once(WEB_ROOT.'server_libraries/recorder.php');

// 1. Initialising
$recorder = new Recorder();
$recorder->setRetentionPeriod(10);

// 2. Adding new events
$doc = new EMMADocument($_REQUEST['emma']);
$data = $doc->findElement('emma:interpretation');
if($data->getAttributes()['emma:medium'] == 'visual')
{
    $X = $data->findElement('X')->getRawValue();
    $Y = $data->findElement('Y')->getRawValue();
    $newevent = new HandEvent($X, $Y);
    $recorder->addEvent($newevent);

    // Detecting matches to see if still hand
    $events = $recorder->getEvents();
    $similar = array();
    foreach(array_reverse($events) as $ev)
    {
        if($ev->equals($newevent))
            $similar[] = $ev;
        else
            break;
    }
    if(count($similar) > 1)
    {
        $_SESSION['last_selected'] = array($newevent->getX(), $newevent-
>getY());
        echo json_encode(
            array(
                'action'    =>  'STILL',
                'X'     =>  $newevent->getX(),
                'Y'     =>  $newevent->getY(),
                'time'  =>  $newevent->getTime()-end($similar)->getTime() -
1
            )
        );
```

```
        exit;
    }
}
elseif($data->getAttributes()['emma:medium'] == 'acoustic')
{
    $event = new VoiceEvent($data->findElement('utterance')-
>getRawValue());
    $recorder->addEvent($event);
}
```

The next thing to do is to check for a voice event in the recorder that respects the EMMA lattice described in the previous section:

```
$model = new EMMADocument(file_get_contents("doc/draw_lattices.xml"));
$lattice = $model->findElement("emma:lattice");

$combo = $recorder->detectCombination(array($lattice));
```

If a combination is detected, meaning that a voice event matches the EMMA lattice, an action can be taken. As said in the problem description, user commands can be classified into two categories: the shape creation and shape shifting. According to Figure 8.8, a shape creation occurs when the sentence starts with “draw” or “create”, while a shape shifting occurs when it starts with “put” or “move”.

```
$result = array();
$utterance = $combo[0];
$words = explode(" ", $utterance->getUtterance());
if(in_array($words[0], array("draw", "create")))
{
    // Shape creation
}
elseif(in_array($words[0], array("put", "move")))
{
    // Shape shifting

}
```

In case of a shape creation, the script will have to handle the optional “a” and size keywords. Default size is small (50 pixels) but can be large (100px) if asked by the user. The result sent back to the client will be a “DRAW” command with needed information encoded as a JSON document:

```
$index = 1;
$size = 50;
if($words[$index] == "a")
    $index++;
if(in_array($words[$index], array("small", "large")))
{
    // Given size
    $size = ($words[$index]=="small"?50:100);
    $index++;
}

if(in_array($words[$index], array("blue", "red", "green")))
```

```
{
    // Shape drawing
    $color = $words[$index];
    $shape = $words[$index+1];
    $result = array(
        'action'    =>  'DRAW',
        'shape'     =>  $shape,
        'color'     =>  $color,
        'size'      =>  $size
    );
}
```

In case of shape shifting, the script needs to extract the two last hand coordinates to detect which object has to be moved and where to move it. Then, as the EMMA lattice specifies, the script needs to extract optional further information such as size, color and shape. Eventually, it will have to send back to the client script a “MOVE” command with information about the object to move and where to move it:

```
$point1 = new HandEvent(HandEvent::ANY_COORDINATE,
HandEvent::ANY_COORDINATE);
$point2 = new HandEvent(HandEvent::ANY_COORDINATE,
HandEvent::ANY_COORDINATE);
$combo = $recorder->detectCombination(array($point1,$point2));
if($combo != null)
{
    if(in_array($words[2], array("there", "here")))
    {
        // No details given
        $size = "";
        $color = "";
        $shape = "";
    }
    else
    {
        $index = 2;
        $size = "";
        if(in_array($words[$index], array("large", "small")))
        {
            $size = ($words[$index] == "large"?100:50);
            $index++;
        }
        $color = $words[$index];
        $shape = $words[$index+1];
    }

    $result = array(
        'action'    =>  'MOVE',
        'from'      =>  array(
                            'X'     =>  $combo[0]->getX(),
                            'Y'     =>  $combo[0]->getY(),
                            'size'  =>  $size,
                            'color' =>  $color,
                            'shape' =>  $shape
        ),
        'to'        =>  array(
                            'X'     =>  $combo[1]->getX(),
```

```
                                'Y'      =>  $combo[1]->getY()
            )
        );
}
```

At this point, the server has finished its process and has sent a JSON-encoded result to the client script. The *onEvent* function of the recorder will be called and has therefore to be implemented. The script will obviously have to detect which kind of command has to be executed. In case of a “DRAW” command the shape will be drawn in the canvas. The position at which the shape will be drawn has to be calculated, as what is received are actually coordinates for the full web page screen. The coordinates have such to be first converted in pixels so that the HTML5 canvas can understand it correctly and then shifted to the left and top. Indeed, the canvas is situated at a certain distance of the page borders. To keep a correspondence between the visual feedback cursor and the place where the shape will be drawn, it is necessary to make these calculations. This is the purpose of the *marginX* and *marginY* variables defined above.

In case of a “STILL” command, the script will have to give a visual feedback to the user mentioning the shape has been selected. This visual feedback is done by drawing a circle around the selected shape.

In case of shape shifiting, the script will have to move the object. As the HTML5 canvas does not support erasing figures, the shape shifting will be implemented by registering in a JavaScript array all shapes, saving new shapes into it and then extracting their properties when they need to be moved. The shifting is actually a combination of erasing the existing shape in the canvas and then creating it again at the destination spot.

```
var response = JSON.parse(result);
if(!(Object.keys(response).length === 0))
{
    if(response.action == "DRAW")
    {
        var X = last_coords[0]/100*$(window).width()-marginX-
(response.size/2);
        var Y = last_coords[1]/100*$(window).height()-marginY-
(response.size/2);
        shapes[shapes.length] = [response.shape, response.color, X, Y,
response.size];
        drawShape(canvas, X, Y, response.shape, response.color,
response.size);
    }
    else if(response.action == "STILL")
    {
        if(response.time > 2)
        {
            // Select form
            var index = nearestShapeIndex(response.X, response.Y);
            var X = shapes[index][2];
```

```
            var Y = shapes[index][3];
            last_selected = [X, Y];
            var size = shapes[index][4]*2;
            var ctx = canvas.getContext('2d');
            ctx.strokeStyle = 'white';
            ctx.beginPath();
            ctx.arc(X+(size/4),Y+(size/4),(size/2),0,Math.PI*2,true);
            ctx.stroke();
        }
    }
    else
    {
        var index = nearestShapeIndex(response.from.X,
                response.from.Y,
                response.from.size,
                response.from.color,
                response.from.shape
        );
        if(index === null)
        {
            alert("No available shape to move according to criteria");
            return;
        }
        var shape = shapes[index];
        var X = shape[2];
        var Y = shape[3];
        var color = shape[1];
        var size = shape[4];
        shape = shape[0];

        // Erase first old location
        drawShape(canvas, X-size/2-1, Y-size/2-1, "square",
canvasBackground, size*2+2);


        // Draw on new location and update in shapes rray
        var X = last_coords[0]/100*$(window).width()-marginX-(size/2);
        var Y = last_coords[1]/100*$(window).height()-marginY-(size/2);
        shapes[index][2] = X;
        shapes[index][3] = Y;
        drawShape(canvas, X, Y, shape, color, size);
    }
}
```

The *drawShape* function handles the drawing in the canvas. The drawing requires a canvas context, on which multiple operations can be done. The setting of color is made by giving a value to the *fillStyle* property. The *fillRect*, *lineTo* and arc methods are standard methods which allow drawing any shape (Mozilla, n.d.). Here below is the code allowing square, triangles and circles creations:

```
var drawShape = function(canvas, X, Y, shape, color, size){
    var ctx = canvas.getContext('2d');
    ctx.fillStyle = color;
    if(shape === "square")
    {
        ctx.fillRect(X, Y, size, size);
```

```
    }
    else if(shape === "triangle")
    {
        ctx.beginPath();
        ctx.moveTo(X+(size/2),Y);
        ctx.lineTo(X,Y+size);
        ctx.lineTo(X+size,Y+size);
        ctx.fill();
    }
    else if(shape === "circle")
    {
        ctx.beginPath();
        ctx.arc(X+(size/2),Y+(size/2),(size/2),0,Math.PI*2,true);
        ctx.fill();
    }
    else
    {
        console.log("unknown shape");
    }
}
```

Finally, the *nearestShapeIndex* function allows finding the nearest shape from the points that are coming from the server result so that request such as “put the large blue triangle there” can be properly handled.

```
var shapes = [];
var nearestShapeIndex = function(X, Y, size, color, shape)
{
    size = size || "";
    color = color || "";
    shape = shape || "";
    var nearest = -1;
    var distance = -1;
    var i;
    for(i=0;i<shapes.length;i++)
    {
        a = Math.abs(shapes[i][2]-X);
        b = Math.abs(shapes[i][3]-Y);
        dist = Math.sqrt((a*a)+(b*b));
        if(distance === -1 || dist < distance)
        {
            console.log(size);
            cond1 = (size=="" || shapes[i][4] === size);
            cond2 = (color == "" || shapes[i][1] === color);
            cond3 = (shape == "" || shapes[i][0] === shape);
            if(cond1 && cond2 && cond3)
            {
                nearest = i;
                distance = dist;
            }
        }
    }
    if(nearest===-1)
        return null;
    else
        return nearest;
```

```
}
```

9. REVIEW

In the previous chapters has been presented a new multimodal framework designed for web applications and a set of examples of usage. These examples demonstrated the capabilities of the multimodal framework introduced in chapter 7. This chapter aims to objectively review the work and choices that have been done in this thesis. Section 9.1 focuses on reviewing the multimodal framework introduced in chapter 7. Section 9.2 discusses the EMMA language and its adequacy to multimodality. Finally, section 9.3 gives a list of suggestions to improve the framework by considering its review of section 9.1.

9.1. LIBRARY REVIEW

The library shows promising results. It has been built to be easily extensible and pluggable into any system supporting multimodality and EMMA parsing. The main asset of this library is the client JavaScript part which offers an easy interface to building WEB multimodal applications. It is documented and illustrates usage with working examples. Moreover, as it sends data to a server in the form of an EMMA document, which is a W3C standard, it is compatible with any system that is able to parse EMMA. The response format of the server is completely free as it is directly passed to the web application-specific script that uses the multimodal library. In this thesis, we have chosen to use JSON but that is only an example. The library is open to configuration and abstracts all complications relative to the technology used to the user. Finally, it has been observed that the library still functions properly for each combination of these modalities, as illustrated in the examples of chapter 8. As a result, it is simple of use, operational and easily deployable to any multimodal web application.

Nevertheless, even if results are promising, the library is probably not fit for professional web applications yet. This is mostly caused by the technologies such as voice and gesture recognition that are not yet performant enough to handle worst-case scenarios. As described in chapter 6, voice is influenced by surrounding noise and accents while gesture recognition is highly influenced by background and light. Even if the framework helps to reduce some of the bugs and problems of the technology, it still may appear as buggy and fragile for those reasons. This can be solved by enhancing technologies, which will probably happen in the next few years, as WEB history has shown that it constantly improves over the years, considering the work of the WEB community over open projects like WebRTC (WebRTC, 2011).

Concerning EMMA support in the framework, it was developed in a “quick and dirty” way. Currently, there is very limited support for EMMA and the architecture of Figure 7.8 was not compatible with XML tools for JavaScript. As a result, EMMA was supported by using operations on strings, like concatenations, string replacements…

This is not a proper way of developing a valid framework. However, this was not the primary goal of this thesis and was therefore half-ignored. A much more important purpose was to evaluate EMMA as a language and its adequacy to multimodality, which is discussed in the next section.

As for the fusion engine of the server library (through the *detectCombination* method of the server recorder class), it is operational but probably not a powerful one. Yet it manages to detect groups of interesting events based on selection criteria, which helps the library user when trying to find a match. For instance, the “put that there” example uses this engine to find a combination of three events: a voice event with the “put that there” utterance and two touch events where the first one is on a moveable object and the second one on a container. However, partly due to the way the google speech recognition API returns its results, it is not possible to intercross data by using time for example. If the user wants to find in the incoming events a voice event with utterance “that” and a touch event happening at the exact same time, the fusion engine will not be able to fulfill this demand. Of course, this can be bypassed as explained in section 8.2 by using criteria on touch. Furthermore, this fusion engine is not able to detect contradictions or ambiguity. This could be included in a future version of the framework that implements some error and ambiguity handling.

Table 9.1 (introduced in chapter 1) illustrates a comparison of the developed framework with other existing frameworks. It takes into account the presence of a fusion engine which is in charge of merging all the events coming from the recognizers to understand the user’s request. The fission engine, which is responsible for deciding how to address the response to the user and split it into multiple modalities if required, is taken into account too. Finally, table 9.1 references the presence of a dialog manager, which is in charge of understanding the user’s request according to context. The framework developed in this thesis will be referenced as *MultiWeb*. The other frameworks are introduced in chapter 1 of this thesis.

Toolkit	Fusion Engine	Dialog Manager	Fission Component	Web Support
ICon				
OpenInterface				
Squidy				
MEngine	✔	✔		
CoGenIVE	✔	✔		
HephaisTK	✔	✔		
PetShop	✔	✔	✔	
Hinckley	✔	✔	✔	

MultiWeb	✔			✔

Table 9.1. Comparison between frameworks of available features. Table is based on (Cuenca, et al., 2014)

It appears that the framework developed in this thesis lacks both a fission engine and a dialog manager. It is however important to notice that that the fusion engine, as described above, is not advanced. It is capable of basic fusion processes such as detecting combinations of events with limited constraints. It is however the only one to be web-oriented. The goal was to do a first attempt at explaining how to integrate multimodality in a web application through an acceptable framework. Considering the fact that this framework offers the support for multimodality, with abstraction of all technical details, that it is highly extensible to other modalities and to other systems capable of parsing EMMA documents, it can be considered as a success.

9.2. EMMA ADEQUACY TO MULTIMODALITY

EMMA was designed for multimodality by the W3C. Its first version was issued in 2003 and has been followed by nine other versions, the latest going back to 2009. Six years later, it still is not broadly used, despite the fact that it is a W3C recommendation. Although it is often mentioned in papers and articles, there does not seem to be much support for it. There are no tools that are capable of parsing EMMA or generating EMMA documents. This whole process is left to the programmer and can be quite time-consuming as it requires a deep understanding of the specification, the process of creating the architecture and finally some time for actual implementation. Furthermore, there are no real examples of usage outside the W3C specification (W3C, 2009). This can be considered as unattractive to programmers.

Concerning the language itself, EMMA is quite descriptive. Its structure is rapidly understood and is self-describing. On the other hand, this expressiveness can cause a lack of performance. Indeed, it may not suit WEB applications dealing with a big amount of data. For instance, putting coordinates of an object in an EMMA document can drastically increase traffic and memory use. Considering the fact that parsing will be required to extract the coordinates as well, it would be much more efficient to just use two double values.

Moreover, EMMA has a great feature called the lattices. They are particularly convenient to define models of speech as state diagrams. The examples of chapter 8 demonstrate how they can be used in a real context. However, using them to formalize user input as explained in the W3C specification requires a complex algorithm. In a system such as the Google speech recognition, the lattice should be generated by the speech recognition itself. Generating the lattice outside the boundaries of the recognizer system would be a problem because the API is a

continuous process. The outcome of this continuous process is a sentence or, more generally, a group of words, which makes it difficult to build on the fly EMMA lattices with multiple paths caused by exclusive interpretations of words. This causes more memory usage and processing on the client's browser, which includes retrieving all possible instances, splitting each one in lists of words and finally fuse them into one lattice. It is therefore more efficient to simply encapsulate the sentences in interpretations tags wrapped in a one-of container. The EMMA specification gives however full freedom on usage, which gives the opportunity for the programmer to act one way or the other according to the situation. Finally, despite the usefulness of those lattices, they lack the capacity of expressing the "any value" concept or more precisely, the "any value of that type" concept. Indeed, there is no proper way of expressing in a lattice that one particular part of the sentence needs to be an integer for example. This is problematic for infinite domains such as numbers, forcing the user to come up with a meta-value meaning "any value of that type".

EMMA supports three different media of input: gesture, acoustic and ink, which actually are the only three the W3C specification allows. Even if currently, there are no really other media input types, it could happen in the future, with the rise of internet of things sensors, that other media input such as olfaction or vibrations emerge. For this particular attribute, EMMA is not really extensible, which is probably a mistake. On the other hand, it constraints the medium type string, forcing the user to write a standard string, which eventually makes the parsing algorithm and process easier.

Finally, it is important to notice that EMMA does not respect all the CARE properties (Complementarity, Assignment, Redundancy, Equivalence) discussed in chapter 1. The complementarity can be achieved by using group containers. The assignment property is naturally there. Redundancy can be implemented with a one-of container and a multi-device or multi-process disjunction type. However, equivalence cannot be expressed, due to the fact that an EMMA document has to include multiple interpretations in a container and that there is not a suitable container capable of expressing equivalence. "One-of" containers describe exclusive disjunction, group containers complementarity and sequence containers just sequences. None of them can express the inclusive disjunction required by the equivalence property.

According to (Dumas, et al., 2010), description languages for multimodality can be evaluated based on this list of guidelines:

1. *Multiple levels of abstraction*: these multiple levels of abstraction are directly related to the multimodal interface modelling. It can for example be useful to abstract certain technical details.

2. *Events management*: this aims to attach meaning to a model. For instance, routines to be executed when a certain combination of modalities has been detected can be specified in the model.
3. *Support for time synchronicity:* for instance, it should be possible to indicate if events are happening in parallel or sequentially.
4. *The adaptation to context and user*: this is mainly a fission feature. The language should be able to specify which modalities to launch in which context.
5. *Error handling*: this describes a set of potential error sources that can be listed and optionally have their respective fallback routines.
6. *Data modelling:* data encapsulation and representation.

The EMMA language offers support for point 3 and 6. EMMA is highly designed for data modelling and encapsulating. Different types of containers, attributes like *emma:source* or *emma:medium* are all designed to encapsulate data and to enrich it with meta information. Time synchronicity can be mildly expressed through the available EMMA containers. The *sequence* container expresses a sequence of events, while event happening in parallel can be expressed by using the other containers. EMMA is in fact mostly designed to represent incoming data but not to be used as configuration files or interfaces models that could influence the working process of the fusion engine. The only exception is for the EMMA lattices. Actually, this is a general problem with EMMA: it is mostly designed for the voice modality. The support for other modalities is less advanced. For instance, there is no feature in EMMA to describe an ink trace like defined in the InkML specification (W3C, 2011).

EMMA seems to be fit for multimodality at first sight and for simple multimodal applications. It is however not really extensible (illustrated by its medium property and the little support of modalities other than voice) and unused. Even after nine reviews of the specification and its 12-year existence, there are no available tools to handle EMMA. Worse, there does not seem to be any actual usage. This is probably due to a combination of its lack of extensibility and available tools. In the light of these arguments, it seems that EMMA is suitable for basic multimodal applications but can be restricting for advanced ones.

9.3. FUTURE DIRECTIONS

The framework is currently usable but could benefit from ameliorations. It only supports voice, touch and hand input modalities. Future versions of this library could include more modalities like face, smile, fist, body movement. These can all be implemented by using the *objectdetect* library. The framework is easy to extend, it just requires adding methods in the client recorder JavaScript “class”. Current recognizers can also be replaced by new technologies or more performant ones. Indeed, even if results seem promising, they are not fit for big applications yet. The

Google speech recognition API will probably improve over time. This is less certain for the *objectdetect* library, which seems to be part of an independent project. More technologies could appear in the future for gesture recognition or even handwriting recognition, which is already available in the translation google tool but not open to the public yet.

Tools for EMMA parsing and generation available in the JavaScript and server libraries could benefit from a big enhancement. They are not currently completely implemented, as the server library lacks EMMA generation methods and the JavaScript library lacks support for lattices and parsing. Moreover, there is probably a better and more optimized way of parsing the documents than the algorithm that was used in both libraries. Indeed, it is based on regexes and string manipulations, while the best way of implementing it would be to use XML existing tools. Those tools were at first tried in this thesis but were not compatible and caused a few problems. Therefore, as this was not the primary goal of this thesis, it was not optimized and finished.

Matching an utterance to an EMMA lattice (state machine) is one of the features of the server library. The algorithm used simply checks for a valid path through the state machine, which is acceptable. However, it does not support non-deterministic state machines and does not use an optimized algorithm such as equations solving and translation to regular expression using the Lemme theorem (Marcozzi, 2014). This is however not a simple algorithm to develop and has thus be considered out-of-topic and left aside.

The server library could also benefit from an enhanced fusion engine. The current one is able to detect a group of events with some constraints like regions of the screen, utterance exact match or match to a lattice. However, there is no support for advanced constraints like the one represented in Figure 1.3 (chapter 1) with data intercrossing. This can be done by extending the PHP library or by using an existing fusion engine and plugging it into a web server. This server would have to support EMMA parsing and could be written in any language of a web engine.

Furthermore, there is currently no fission engine. In multimodality, fission is the process that decides which form to apply to the response or a given message depending on context. For example, it would be wise not to use speech synthesis in noisy areas. Fission has not been addressed in this thesis and has not been taken into account in the multimodal framework. This is however easily embeddable in the framework thanks to its genericity. Indeed, as the server response is completely free, a new layer of abstraction could be built on top of the current framework, with a standard communication format and JavaScript functions to abstract the parsing of the response.

Finally, there is currently only support for the server framework in PHP. It would be more convenient to users if there were multiple engines and framework tools written in different languages, so that they could be used by a wider group of programmers and web applications. Moreover, writing the server library in different languages would be an opportunity of comparing which one handles the best fusion engines and event handling based on several parameters like robustness, processing rapidity or language capacities.

CONCLUSION

This thesis has shown that multimodality is indeed possible in web applications. Nevertheless, it has brought up a few problems that are mostly related to the unreliability of recognition engines, such as voice and gesture. Although the results observed in the web multimodal applications of chapter 8 seem promising, they are not fit for real-life applications yet. The context in which the web application is executed influences a lot the behavior of the web application. However, this framework is only a starting point. As detailed in section 9.3, the framework is quite generic and open to ameliorations. For instance, a fission engine built on top of this library could be easily added and bring modularity to web applications. This would make it possible to easily switch between modalities depending on the context in which they are executed.

In chapter 1 has been introduced the concept of multimodality and the benefits of multimodality in user interfaces. It has also introduced a list of existing frameworks that handle multimodality and their benefits. However, none of them were designed for multimodality. It has concluded that multimodality could be added to web applications by using the new features of HTML5.

Chapter 2 has introduced the voice modality and how it could be implemented in a web browser. Voice recognition is available through the webkitSpeechRecognition Google API, while voice synthesis can be achieved by the speechSynthesis API. These libraries are promising but can be considered as unreliable. Although the speech recognition API is operational, it still requires a proper accent and a silent environment. Speech synthesis is better, meaning that it can be used in a real-life web application, but is sadly instantiated by the web browser, resulting in lack of control from the web developer. The chapter ended with a list of parameters to take into account if rigorous tests had to be conducted for these libraries.

In chapter 3, the availability of the gesture modality in web applications has been discussed. It has determined that touch was easily achievable. Pointing could not be implemented directly because it requires hardware capable of generating three-dimensional data. However, pointing can be partially achieved by using gesture recognition with the *objectdetect* library, which has shown some promising results. It is built over the WebRTC API and is able to detect body parts on data recorded by the webcam, like hands, fists, eyes or mouth. It has some defaults and is not completely reliable, making it unfit for real-life applications. However, it was kept for further use to answer the needs of this thesis. This chapter concluded by proposing a set of rigorous tests to be conducted on the *objectdetect* library to evaluate its performance.

The ink modality was approached in chapter 4. The ink modality is a communication channel that takes place through a pen or a stylus. It was possible to include it in web applications by using the HTML5 canvas element and an ink recognition engine. Tests of the engines that were found have sadly demonstrated that there was no acceptable technical solution to implement handwriting recognition in web browsers. Indeed, OCR engines, which are capable of recognizing handwritten text, have all failed to recognize the handwritten text samples that were used as inputs. The $P algorithm, which is able to recognize any form (drawing, letter…) by trying to match the drawn input to a *model*, has shown interesting results. However, the set of pre-defined models was too small to be used directly and would have required substantial work to be really operational. For these multiple reasons, the ink modality was abandoned.

In chapter 5, the EMMA language was introduced. EMMA is a multimodal annotation language recommended by the W3C. Its structure has been described at first, followed by a detailed description of the most interesting elements. An early review has been done in section 5.3 that concluded that EMMA was at first sight suitable for multimodality.

A summary and global overview has been presented in chapter 6. A comparative table has compared all libraries found in chapters 2, 3 and 4 and assessed them on multiple criteria. The first criterion was the compatibility of browsers. It appeared that only Google Chrome has support for all libraries, with Opera coming as a close second. The performance of all libraries has also been included in the criteria, by assessing two factors: their operability and direct usability in real-life web applications. The ease of programming has also been taken into account as well as their configurability. Finally, the comparative table mentioned the compatibility with living standards and its open source factor. Given this overview, a set of libraries have been selected to be carried out further in this thesis, while others were abandoned, like ink recognition.

The second part of the thesis, beginning with chapter 7, aimed to develop a multimodal framework fit for web applications. The development has been a bottom-up process, meaning that the library was gradually completed by implementing the examples of chapter 8. The global architecture was however decided before the implementation began in order to keep a clean global structure. The framework has been split into two parts. The first part is written in JavaScript and is meant to be executed on the client's browser. This part of the framework is responsible for starting the recognizers and adding a level of abstraction over the libraries of chapters 2 and 3. This level of abstraction has resolved some issues coming from the raw recognizers. The second part of the framework was the server library written in PHP. As opposed to the client's part of this framework, which acts as an event *listener*, the server library acts as an event *processor.* Its responsibility is to record all

incoming events and merge them to understand the user's intent. This understanding is the responsibility of the fusion engine.

The framework uses EMMA for communication purposes between the client's browser and the server. Both ends have support for EMMA parsing and generation, with a structure based on the composite pattern. It is also able to handle EMMA lattices, which are mathematical graphs with named arcs, convert them into in-memory state machines and finally match a list of words (typically a sentence) to this state machine.

Chapter 8 has presented a set of four examples that illustrate the capabilities of the multimodal framework. These examples have also been a way of gradually building the framework. This process has brought up a list of problems that were resolved as they came up. For example, it appeared that the coordinates sent back by the *objectedetect* library that was used to handle the gesture modality were not always relevant. On the subject of the voice modality, it was observed that it generated two types of results: interim results, which were faster but less reliable and final results, which were slower but more reliable. This problem was a real dilemma but was partially resolved.

The first example of chapter 8 aimed to implement a simple "put-that-there" application by combining the voice and touch modalities. In this example, the user could move an object across the screen by touching it, uttering the "put that there" sentence and touching the destination area. This made it possible to develop as well the touch and voice modalities of the framework.

The second example aimed to create a web application where the user could move the cursor and select an object by waving at the camera. The *objectdetect* library was used for this purpose with its hand recognizer. This example introduced the gesture modality in the multimodal framework.

In the third example, the stress was put on a small dialog manager that could be achieved with EMMA lattices. The web application is able to understand a small set of sentences in natural language to find flights from one airport to another. For instance, uttering "flights from Boston to Austin" is understood by the web application, which then lists all available flights in a table. This example served as a basis to implement the part of the framework that could convert an EMMA lattice into an in-memory state machine and match the voice input to this state machine.

The fourth example can be seen as a combination of all the previous examples. It aimed to implement a revisited "put-that-there" web application as close as possible to Bolt's experiment (Bolt, 1979). It combined the voice and gesture modalities to create colored shapes on the screen, which could then be moved. The user could

utter a wide set of sentences that were defined in the EMMA lattice and that the web application was capable of understanding.

In the last chapter was discussed a review of the framework presented in the second part of this thesis. It was concluded that it was not as advanced as the other existing frameworks presented in chapter 1 as it lacked a fission engine and a dialog manager. However, it was the only one that offered web support. Some features like the EMMA algorithms or the fusion engine can benefit from a big enhancement. A list of future directions has been given in section 9.3 to improve the framework: a better algorithm for EMMA lattices matching, the development of a fission engine and the implementation of the server library in multiple languages other than PHP to extend support. The framework is however only a starting point. The goal was to do a first attempt of multimodality integration in web development, which has been a success regarding the key points of the library presented in section 9.1.

Finally, the last chapter has discussed the adequacy of EMMA to multimodality. It has concluded that even if it seems adequate to multimodality at first, it can rapidly become restricting for advanced multimodal frameworks. It is quite focused on the voice modality but limited in other modalities like ink. Its lack of usage, despite its standardization by the W3C and its 12-year existence, can be considered as unattractive, due to the required substantial work to develop tools for EMMA support. In the light of these observations, one could ask about the future of EMMA, whether or not its usage will expand with the potential arrival of multimodality in web applications and if EMMA can be enhanced to support advanced multimodality features.

REFERENCES

[1] ABBYY. Abbyy finereader. http://www.abbyy.com/finereader. (Accessed in 2015).

[2] Adobe. Adobe acrobat DC. https://www.acrobat.com. (Accessed in 2015).

[3] Eric Bidelman. Capturing audio & video in HTML5. http://www.html5rocks.com/en/tutorials/getusermedia/intro/. (Last updated on 29-Oct-2013) (Accessed in 2015).

[4] Eric Bidelman. Web apps that talk - introduction to the speech synthesis API. http://updates.html5rocks.com/2014/01/Web-apps-that-talk---Introduction-to-the-Speech-Synthesis-API. (Last updated on 14-Jan-2014) (Accessed in 2015).

[5] Richard Bolt. Put that there (original). https://www.youtube.com/watch?v=RyBEUyEtxQo. (Posted online 25-oct-2006) (Accessed in 2015).

[6] Richard A Bolt. “Put-that-there”: Voice and gesture at the graphics interface, volume 14. ACM, 1980.

[7] Marie-Luce Bourguet. A toolkit for creating and testing multimodal interface designs. In *Proceedings* of UIST’02, pages 29–30, 2002.

[8] Thomas Breuel. Announcing the OCRopus open source OCR system. http://googlecode.blogspot.be/2007/04/announcing-ocropus-open-source-ocr.html. (Posted online on 09-apr-2007) (Accessed in 2015).

[9] Fredy Cuenca. The Cogenive concept revisited: A toolkit for prototyping multi-modal systems. In *Proceedings of the 5th ACM SIGCHI symposium on Engi-neering interactive computing systems*, pages 159–162. ACM, 2013.

[10] Fredy Cuenca, Karin Coninx, Davy Vanacken, and Kris Luyten. Graphical toolkits for rapid prototyping of multimodal systems: A survey. *Interacting with Computers*, page iwu003, 2014.

[11] Sébastien De la Marck and Johann Pardanaud. Les objets. In *Dynamisez vos sites WEB avec JavaScript*, pages 219–234. Simple IT, Condé-sur-Noireau, France, 2012.

[12] Ian Devlin. Using the WEB speech API to control a html5 video. http://www.iandevlin.com/blog/2014/01/javascript/using-the-web-speech-api-to-control-a-html5-video. (Posted online on 03-jan-2014) (Accessed in 2015).

[13] Bruno Dumas, Denis Lalanne, and Rolf Ingold. Hephaistk: a toolkit for rapid prototyping of multimodal interfaces. In *Proceedings of the 2009 international conference on Multimodal interfaces*, pages 231–232. ACM, 2009.

[14] Bruno Dumas, Denis Lalanne, and Rolf Ingold. Description languages for multimodal interaction: a set of guidelines and its illustration with SMUIML. *Journal on multimodal user interfaces*, 3(3):237–247, 2010.

[15] Bruno Dumas, Denis Lalanne, and Sharon Oviatt. Multimodal interfaces: A survey of principles, models and frameworks. In *Human Machine Interaction*, pages 3–26. Springer, 2009.

[15] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[16] Gardnerp. Html 5 canvas - a simple paint program (touch and mouse). http://www.codeproject.com/Articles/355230/HTML-Canvas-A-Simple-Paint-Program-Touch-and-Mou. (Posted online on 26-apr-2012) (Accessed in 2015).

[17] Gilmoreorless. Jquery-nearest. https://github.com/gilmoreorless/jquery-nearest. (Last updated on 26-apr-2015) (Accessed in 2015).

[19] Google. Web speech api creates interactive experiences.

http://commondatastorage.googleapis.com (Accessed in 2015)

[20] Google. Web speech api demonstration.

https://www.google.com/intl/en/chrome/demos/speech.html. (Accessed in 2015).

[21] Alex Graves, Marcus Liwicki, Santiago Fernandez, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions* on, 31(5):855–868, 2009.

[22] Alfred Haar. Zur theorie der orthogonalen funktionensysteme*. Mathematische*

Annalen, 71(1):38–53, 1911.

[23] Hakimel. The html presentation framework. https://github.com/hakimel/reveal.js. (last update 18-may-2015) (Accessed in 2015).

[24] Ken Hinckley, Mary Czerwinski, and Mike Sinclair. Interaction and modeling techniques for desktop two-handed input. In *Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 49–58. ACM, 1998.

[25] Lode Hoste, Bruno Dumas, and Beat Signer. Mudra: a unified multimodal interaction framework. In *Proceedings of the 13th international conference on multi-modal interfaces*, pages 97–104. ACM, 2011.

[26] Werner A König, Roman Rädle, and Harald Reiterer. Interactive design of multimodal user interfaces. *Journal on Multimodal User Interfaces*, 3(3):197–213, 2010.

[27] Kripken. Emscripten. https://github.com/kripken/emscripten. (Last updated on 28-apr-2015) (Accessed in 2015).

[28] Kevin Kwok. Ocrad.js. https://antimatter15.com/2013/12/ocrad-js-pure-javascript-ocr-via-emscripten/. (Posted online on 31-dec-2013) (Accessed in 2015).

[29] Michaël Marcozzi. Construction d'une expression régulière pour le langage d'un automate fini. In *Théorie des Langages de Programmation : Syntaxe et Sémantique: Eléments théoriques et exercices*, page 11. Presses Universitaires de Namur, Namur, Belgique, 2014.

[29] Mozilla. Drawing shapes with canvas. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Drawing_shapes. (Accessed in 2015).

[30] Mtschirs. js-objectdetect. https://github.com/mtschirs/js-objectdetect/. (Last updated on 20-feb-2015) (Accessed in 2015).

[31] David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Transactions on*

Computer-Human Interaction (TOCHI), 16(4):18, 2009.

[32] Nicomsoft. OCR SDK. http://www.nicomsoft.com. (Accessed in 2015).

[33] George Ornbo. The HTML5 speech recogntion api. http://shapeshed.com/html5-speech-recognition-api/. (Posted online on 21-jan-2014) (Accessed in 2015).

[34] Sharon Oviatt. Multimodal interfaces. *The human-computer interaction handbook: Fundamentals, evolving technologies and emerging applications,* pages 286–304, 2003.

[35] Jussi Pakkanen. Linux port of Cuneiform. https://launchpad.net/cuneiform-linux. (Last updated on 19-apr-2011) (Accessed in 2015).

[36] Joerg Schulenburg. GOCR. http://jocr.sourceforge.net. (Last updated on 05-mar-2013) (Accessed in 2015).

[37] Marcos Serrano, Laurence Nigay, Jean-Yves L Lawson, Andrew Ramsay, Roderick Murray-Smith, and Sebastian Denef. The Openinterface framework: a tool for multimodal interaction. In *CHI'08 Extended abstracts on human factors in computing systems*, pages 3501–3506. ACM, 2008.

[38] Ray Smith. Tesseract-ocr. https://code.google.com/p/tesseract-ocr/. (Last updated on apr-2013) (Accessed in 2015).

[39] Ray Smith. An overview of the tesseract OCR engine. In *ICDAR*, volume 7, pages 629–633, 2007.

[40] Tmbdev. Python-based OCR package using recurrent neural networks. https://github.com/tmbdev/ocropy. (Last updated on 24-apr-2015) (Accessed in 2015).

[42] TurboSquid. Falling pixel hand. http://www.fallingpixel.com/products/6592/mains/hand01.jpg. (Accessed in 2015).

[43] Ubuntu. Reconnaissance optique de caractères (roc). http://doc.ubuntu-fr.org/ocr. (Accessed in 2015).

[44] Davy Vanacken, Joan De Boeck, Chris Raymaekers, and Karin Coninx. Nimmit: A notation for modeling multimodal interaction techniques. In *GRAPP 2006: Proceedings of the First International Conference on Computer Graphics Theory and Applications,* pages 224–231, 2006.

[45] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O Wobbrock. Gestures as point clouds: a $P recognizer for user interface prototypes. In *Proceedings of the 14th ACM international conference on Multimodal interacti*on, pages 273–280. ACM, 2012.

[44] Paul Viola and Michael J Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

[45] W3C. EMMA: Extensible multimodal annotation markup language. http://www.w3.org/TR/emma. (Last updated on 10-Feb-2009) (Accessed in 2015).

[46] W3C. Ink markup language (InkML). http://www.w3.org/TR/InkML/. (Last updated on 20-Sep-2011) (Accessed in 2015).

[47] W3C. Media capture and streams. http://www.w3.org/TR/mediacapture-streams/. (Last updated on 14-Apr-2015) (Accessed in 2015).

[48] W3C. Multimodal interaction activity. http://www.w3.org/2002/mmi/. (Posted online on 22-May-2014) (Accessed in 2015).

[49] W3C. Web speech API specification. https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html. (Last updated on 19-Oct-2012) (Accessed in 2015).

[50] WebRTC. Webrtc architecture. http://www.webrtc.org/architecture. (Accessed in 2015).

[51] Willy-vvu. reveal-js. https://github.com/willy-vvu/reveal.js. (Last updated on 18-may-2015) (Accessed in 2015).

APPENDIX

1. SOURCE CODE

The source code of the web multimodal framework, as well as the examples given in chapter 8 can be found on the CD enclosed to this thesis. This code requires Google Chrome and an HTTP server capable of interpreting PHP.

2. W3C SPEECH RECOGNITION SPECIFICATION

```
[Constructor]
    interface SpeechRecognition : EventTarget {
        // recognition parameters
        attribute SpeechGrammarList grammars;
        attribute DOMString lang;
        attribute boolean continuous;
        attribute boolean interimResults;
        attribute unsigned long maxAlternatives;
        attribute DOMString serviceURI;

        // methods to drive the speech interaction
        void start();
        void stop();
        void abort();

        // event methods
        attribute EventHandler onaudiostart;
        attribute EventHandler onsoundstart;
        attribute EventHandler onspeechstart;
```

```
    attribute EventHandler onspeechend;
    attribute EventHandler onsoundend;
    attribute EventHandler onaudioend;
    attribute EventHandler onresult;
    attribute EventHandler onnomatch;
    attribute EventHandler onerror;
    attribute EventHandler onstart;
    attribute EventHandler onend;
};

interface SpeechRecognitionError : Event {
    enum ErrorCode {
      "no-speech",
      "aborted",
      "audio-capture",
      "network",
      "not-allowed",
      "service-not-allowed",
      "bad-grammar",
      "language-not-supported"
    };

    readonly attribute ErrorCode error;
    readonly attribute DOMString message;
};

// Item in N-best list
```

```
    interface SpeechRecognitionAlternative {
        readonly attribute DOMString transcript;
        readonly attribute float confidence;
    };

    // A complete one-shot simple response
    interface SpeechRecognitionResult {
        readonly attribute unsigned long length;
        getter SpeechRecognitionAlternative item(in unsigned long
index);
        readonly attribute boolean final;
    };

    // A collection of responses (used in continuous mode)
    interface SpeechRecognitionResultList {
        readonly attribute unsigned long length;
        getter SpeechRecognitionResult item(in unsigned long index);
    };

    // A full response, which could be interim or final, part
of a continuous response or not
    interface SpeechRecognitionEvent : Event {
        readonly attribute unsigned long resultIndex;
        readonly attribute SpeechRecognitionResultList results;
        readonly attribute any interpretation;
        readonly attribute Document emma;
    };
```

```
// The object representing a speech grammar
[Constructor]
interface SpeechGrammar {
    attribute DOMString src;
    attribute float weight;
};

// The object representing a speech grammar collection
[Constructor]
interface SpeechGrammarList {
    readonly attribute unsigned long length;
    getter SpeechGrammar item(in unsigned long index);
    void addFromURI(in DOMString src,
                    optional float weight);
    void addFromString(in DOMString string,
                    optional float weight);
};
```

3. W3C SPEECH SYNTHESIS API

```
interface SpeechSynthesis {
    readonly attribute boolean pending;
    readonly attribute boolean speaking;
    readonly attribute boolean paused;

    void speak(SpeechSynthesisUtterance utterance);
    void cancel();
```

```
  void pause();
  void resume();
  SpeechSynthesisVoiceList getVoices();
};

[NoInterfaceObject]
interface SpeechSynthesisGetter
{
  readonly attribute SpeechSynthesis speechSynthesis;
};

Window implements SpeechSynthesisGetter;

[Constructor,
 Constructor(DOMString text)]
interface SpeechSynthesisUtterance : EventTarget {
  attribute DOMString text;
  attribute DOMString lang;
  attribute DOMString voiceURI;
  attribute float volume;
  attribute float rate;
  attribute float pitch;

  attribute EventHandler onstart;
  attribute EventHandler onend;
  attribute EventHandler onerror;
  attribute EventHandler onpause;
```

```
    attribute EventHandler onresume;
    attribute EventHandler onmark;
    attribute EventHandler onboundary;
};

interface SpeechSynthesisEvent : Event {
    readonly attribute unsigned long charIndex;
    readonly attribute float elapsedTime;
    readonly attribute DOMString name;
};

interface SpeechSynthesisVoice {
  readonly attribute DOMString voiceURI;
  readonly attribute DOMString name;
  readonly attribute DOMString lang;
  readonly attribute boolean localService;
  readonly attribute boolean default;
};

interface SpeechSynthesisVoiceList {
  readonly attribute unsigned long length;
  getter SpeechSynthesisVoice item(in unsigned long index);
}
```

4. PAINTING SCRIPT FOR CANVAS ELEMENT

```
$(document).ready(function () {
        initialize();
    });
```

```
      // works out the X, Y position of the click inside the canvas from
the X, Y position on the page
      function getPosition(mouseEvent, sigCanvas) {
         var x, y;
         if (mouseEvent.pageX != undefined && mouseEvent.pageY !=
undefined) {
            x = mouseEvent.pageX;
            y = mouseEvent.pageY;
         } else {
            x = mouseEvent.clientX + document.body.scrollLeft +
document.documentElement.scrollLeft;
            y = mouseEvent.clientY + document.body.scrollTop +
document.documentElement.scrollTop;
         }

         return { X: x - sigCanvas.offsetLeft, Y: y - sigCanvas.offsetTop
};
      }

      function initialize() {
         // get references to the canvas element as well as the 2D drawing
context
         var sigCanvas = document.getElementById("canvasSignature");
         var context = sigCanvas.getContext("2d");
         context.strokeStyle = 'Black';

         // This will be defined on a TOUCH device such as iPad or Android,
etc.
         var is_touch_device = 'ontouchstart' in document.documentElement;

         if (is_touch_device) {
            // create a drawer which tracks touch movements
            var drawer = {
               isDrawing: false,
               touchstart: function (coors) {
                  context.beginPath();
                  context.moveTo(coors.x, coors.y);
                  this.isDrawing = true;
               },
               touchmove: function (coors) {
                  if (this.isDrawing) {
                     context.lineTo(coors.x, coors.y);
                     context.stroke();
                  }
               },
               touchend: function (coors) {
                  if (this.isDrawing) {
                     this.touchmove(coors);
                     this.isDrawing = false;
                  }
               }
            };

            // create a function to pass touch events and coordinates to
drawer
            function draw(event) {
```

```
                // get the touch coordinates.  Using the first touch in case
of multi-touch
                var coors = {
                    x: event.targetTouches[0].pageX,
                    y: event.targetTouches[0].pageY
                };

                // Now we need to get the offset of the canvas location
                var obj = sigCanvas;

                if (obj.offsetParent) {
                    // Every time we find a new object, we add its offsetLeft
and offsetTop to curleft and curtop.
                    do {
                        coors.x -= obj.offsetLeft;
                        coors.y -= obj.offsetTop;
                    }
                    // The while loop can be "while (obj = obj.offsetParent)"
only, which does return null
                    // when null is passed back, but that creates a warning
in some editors (i.e. VS2010).
                    while ((obj = obj.offsetParent) != null);
                }

                // pass the coordinates to the appropriate handler
                drawer[event.type](coors);
            }


            // attach the touchstart, touchmove, touchend event listeners.
            sigCanvas.addEventListener('touchstart', draw, false);
            sigCanvas.addEventListener('touchmove', draw, false);
            sigCanvas.addEventListener('touchend', draw, false);

            // prevent elastic scrolling
            sigCanvas.addEventListener('touchmove', function (event) {
                event.preventDefault();
            }, false);
        }
        else {

            // start drawing when the mousedown event fires, and attach
handlers to
            // draw a line to wherever the mouse moves to
            $("#canvasSignature").mousedown(function (mouseEvent) {
                var position = getPosition(mouseEvent, sigCanvas);

                context.moveTo(position.X, position.Y);
                context.beginPath();

                // attach event handlers
                $(this).mousemove(function (mouseEvent) {
                    drawLine(mouseEvent, sigCanvas, context);
                }).mouseup(function (mouseEvent) {
                    finishDrawing(mouseEvent, sigCanvas, context);
                }).mouseout(function (mouseEvent) {
                    finishDrawing(mouseEvent, sigCanvas, context);
                });
```

```
            });

        }
    }

    // draws a line to the x and y coordinates of the mouse event inside
    // the specified element using the specified context
    function drawLine(mouseEvent, sigCanvas, context) {

        var position = getPosition(mouseEvent, sigCanvas);

        context.lineTo(position.X, position.Y);
        context.stroke();
    }

    // draws a line from the last coordiantes in the path to the
finishing
    // coordinates and unbind any event handlers which need to be
preceded
    // by the mouse down event
    function finishDrawing(mouseEvent, sigCanvas, context) {
        // draw the line to the finishing coordinates
        drawLine(mouseEvent, sigCanvas, context);

        context.closePath();

        // unbind any events which could draw
        $(sigCanvas).unbind("mousemove")
                    .unbind("mouseup")
                    .unbind("mouseout");
    }
```

5. COLORED THESIS

A colored version of this thesis can be found on the appendix CD.