In this document, I am going to give a detailed description of my semester project in the CHILI lab. First, let us introduce the subject. The application we were developing during these three months is designed to be used by schools for doing some games in order to help students learn mathematics in an easier and funnier way. Actually it is simply a computer vision web browser for now. There is a window and inside that window there is a web browser and a video output on top of it. The idea of the project is to go to a particular web page without writing the url. For example, if we want to go to <https://www.youtube.com/>, we do not have to write the url address for that but we simply have to show the corresponding “Chilitag” to the camera. Chilitags are tags which are similar to the qr codes and which can be detected by the camera. For example, there is a particular chilitag for going to Youtube. If we show that chilitag to the camera it will be detected and the app will look into a database and see that that chilitag is binded to <https://www.youtube.com/>. So it will load that address in the browser.

The idea of the “game” mentioned above is the following: There are groups of four students in the class. Each group has a task. There is an empty geometric area and there is a geometric piece. The students have to use that geometric piece like a puzzle piece and cover as much area as possible in the empty geometric area. It is called tessellation – the tiling of a plane using one or more (one in this case) geometric shapes, called gaps, with no overlaps and no gaps (this definition is from wikipedia). In order to make it funnier, there is a story behind it: someone has broken the Town Hall windows and we need “designers” and “workers” for repairing it. Designers have to analyse the problem and see what is the best way of filling the hole and to write a text document containing the procedure. Then the workers will read that procedure, follow it and try to reconstruct the window in Geogebra. At the end, the teacher will measure the area which is covered and give a number of points to the groups. Thus the group which has left the least “empty space” will have the most points and win the game.

For writing the text document, designers will use google documents which are linked to a particular url. That is an easy way for handling that document. We can imagine that the document is “encoded” in the tag. We show that tag to the app and it displays it in Google docs so that we can read it or write it. So it is not difficult for the designers to give that document to the workers. Moreover chilitags are useful for other things. For example students have to search for videos in Youtube which explain how to do tessellation in Geogebra. But maybe they won't find the best videos. And the teacher has some tags which are linked to helpful videos. The students can pay some points so that the teacher give them those tags. The teacher can also use those tags to pause the game if she wants to make an announcement. See the documents for more information about the game.

The general structure of the application

The application was coded in C++ using Qt/QML. Qt is a popular cross-platform library used for creating Window-based applications <http://www.qt.io/> and QML is a Javascript-based language used to make this task easier. The CHILI lab has developed those chilitags <https://github.com/chili-epfl/chilitags> which already have the Computer Vision functionalities needed for this project. This chilitags library use OpenCV which is an open source library used for Computer Vision <http://opencv.org/> . And there is also qml-chilitags <https://github.com/chili-epfl/qml-chilitags> which contains some functionalities which allow us to use chilitags in QML.

The application is made of the following files: bachelorProject.pro, fileio.h, fileio.cpp, main.cpp, ChilitagsWebsite.qml and main.qml.

Main.cpp

This is the file containing the main method which contains the code to be run. First there are include statements which allow us to use the necessary Qt/QML functions for this application. We also have to include fileio.h which is used for reading information about the tags from a text file. Then, before the main method, there is the Q\_DECL\_EXPORT declaration which must be added to the declarations of symbols used when compiling a shared library. Then there is the main method which contains the declaration of the application as a QGuiApplication instance and the engine as a QQmlApplication engine instance. This engine is loaded with the main.qml and it is also linked to the FileIO class so that FileIO can be used as a QML component in the QML files. Then the application is simply run with the last line 'return app.exec()'.

Fileio.h

This header file contains the declaration of the FileIO class which contains the functionalities needed for reading a file. It extends QObject because we want to use it as a QML component. It contains the methods save and read. The save method is not important for now as the application doesn't write anything from files. The read method will be used to read information from the corresponding text file “tags.txt”. The other two methods are the standart constructor FileIO() and the standard destructor ~*FileIO*() needed in every C++ class.

fileio.cpp

This is the implementation file for the FileIO class. It contains the bodies of the FileIO class methods. First we include the necessary file input and output classes: QFile and QTextStream. We also include fileio.h and QObject. The we have the FileIO's constructor. We don't need to put anything inside since we don't really need instances of FileIO in the qml files. For example, if we want to call the method read, we will just write FileIO.read() and not FileIO fileIO = new …; and then fileIO.read().

Then we have the save method which is useless for now since we don't write into the tags.txt file. But if we want to change the application and to add this functionality, for example if we want to have a system for dynamically linking tags to urls, then we could use it. We create an instance of QFile which we ling to the string “tags.txt”. The file tags.txt is supposed to be located in the same folder as the bachelorProject.pro file. Otherwise the application won't work. We make an if test in order to see whether the file was opened correctly or not. If it is the case we use a text stream to write into the file the same way we would use cout to write into the console. For example, for the console, we would have written 'cout << text << endl' but here we write stream << text << endl. That is simply the way file writing and reading works in C++.

After that, we have the read method which is used, as its name suggests, to read some information from the tags.txt file. We first create an instance of Qfile which we link to the “tags.txt” text file which is supposed to be in the same directory as bachelorProject.pro. Then we declare a QString variable called fileContent. This fileContent variable will be returned at the end of the method if there were no problems with file reading. Then we do a test in order to see whether the file was correctly opened. If it is the case then we read the file: we declare a Qstring instance called line which will be helpful for reading the content of the file line by line. We also declare a text stream t which can be considered as the link between the text file and the string we return at the end. After that we enter a do-while loop until the end of the file is reached (line.isNull()). We read the content of the file line by line with the readLine() method and we also append a '\n' character at the end of each line. After the loop we close the file with the close() method. If the file wasn't opened correctly then we simply return an empty string. At the end of the method, in case there were no reading problems, we can return the fileContent string which now contains the text which is in the tags.txt file.

Finally, the last method of this file is the destructor for the FileIO class. Destructors are used in C++ when we want to free some memory we allocated before for example. More generally, they are used when we want to do some “cleaning up”. In this case, for the FileIO class, we don't need to do anything.

ChilitagsWebsite.qml

In this file we create the qml component ChilitagsWebsite which will be used to easily bind a url to a tag. It contains the logic for a url-tag binding. First we have the import statements. We have to import the QtQuick necessary components. And then we mustn't forget to import Chilitags, QtWebKit for the web browser part and also QtMultimedia for the camera part.

So the ChilitagsWebsite is a ChilitagsObject. ChilitagsObject (from qml-chilitags) has the name property which is an identifier for the tag. Every tag has an id and, if we want to link the ChilitagsObject to the tag number 52 for example, the name property will be name : “tag\_52”. More generally, if we want to link tag number ID to a ChilitagsObject component, then the name property of that component will be “tag\_ID”. ID can be 0, 1, 2, 3, etc. ChilitagsObject also has the visible boolean property which will be true, as its name suggests, when the corresponding tag becomes visible on the screen. We will cover that visible property more in details later. Notice that the name property is “tag\_32” by default. The application is supposed to change that anyway but we could also leave name: “tag\_0” or name: “”.

Then we define some properties that ChilitagsObject doesn't have but that ChilitagsWebsite needs in order to store the url and the message we want to display for a given tag. So we have the webSiteUrl and messageToDisplay string properties. The webSiteUrl property corresponds to the url the tag is linked to. For example, if webSiteUrl is <https://www.youtube.com/> and name is “tag\_32” then showing tag number 32 will load Youtube web page. By default, the value of webSiteUrl is "<https://www.youtube.com/>" but we could leave something like webSiteUrl : “”. It doesn't really matter as long as the application assigns tags to urls at runtime. We also have the messageToDisplay string property. It represents something like more “human-readable” name for the website we are loading. For example, instead of displaying “Going to <https://www.youtube.com/>”, it could be better and more easy for the user to show “Going to Youtube”.

Then we have the timeLoading integer property. Actually we don't want to start loading a web page at the same moment when we show the tag. The reason for that is that the user could unintentionally show a tag to the camera and the web page loading would be started without the user wanting it. The timeLoading property indicates the number of miliseconds between the moment we show the tag and the moment the application starts loading the web page. By default it is 2000, which corresponds to 2 seconds. Thus if a user continuously shows a tag during two seconds then the web page will be loaded.

Then we have the messageLoading and messageReady string properties. Those are the messages that will be displayed when the tag is shown. When it is shown but the loading has not started yet the message will be “Preparing for “ + messageToDisplay. But after timeLoading miliseconds (after 2 seconds by default), the application will display the message “Going to “ + messageToDisplay. The last two properties are the integer counter and the boolean counterIncrementing. Those two properties will be useful for keeping track of how much time has passed since we started showing the tag to the camera.

After the properties we have the onVisibilityChanged “method” which is called every time the visible property changes (from true to false or from false to true). The visible property changes automatically according to whether we are showing the tag to the camera or not. If the tag is now visible, and if the application is not paused, we start preparing for the web page loading. Let's assume that the tag has the name “tag\_32” and it is linked to Youtube. Then we will first write the message “Preparing for Youtube” next to the tag position on the screen. With that we also increase the video output size so that we can better see where the tag is. For increasing the video output size we call the javascript method increaseVideoOutputSize() implemented in the main.qml file. That method will be explained later. And we also start counting by putting the value of counter to 0 and the value of counterIncrementing to true. This will allow the body of the timer's onTriggered method to start being called periodically. In the main.qml file we defined a counterSteps variable. That variable represents the number of periods there are in timeLoading (2 seconds by default). For example, if timeLoading is 2 seconds and counterSteps is 8 then the counter will be increased 8 times every 0.25 second. That is because we want to offer visual feedback to the user. When we show a tag a red circle appears to confirm that the tag has been detected. But if we keep that tag visible to the camera then we see a green circle which grows inside the red one. It grows periodically every timeLoading / counterSteps miliseconds (0.25 seconds in this case). When the size of the green circle matches the red one's size(when both circle have the same size) that means the tag was visible to the camera during 2 seconds so the web page loading can start. That is the reason why we need those counters.

Let go back to our if branching and see what the else branch does. Basically it is called when the tag is no more visible, when the visible property of the tag changes from true to false. It increases the browser size and decreases the video output size because we don't really need the camera when we are not showing tags. We also set the counter value to 0 and the counterIncrementing value to false. Otherwise the program would continue counting time and growing the green circle.

Then we have a Timer component which will be used to see how much time the tag was shown to the camera. The interval property indicates what is the interval between two timer triggers. In our case that will be equal to timeLoading / counterSteps which would be 0.25 sec. So the onTriggered method of this timer will be called every 0.25 seconds. We also have to set repeat and running attributes to true if we want this to work correctly. The onTriggered method contains an if statement which checks whether we have to count now. We have to count if a tag is being shown to the camera and the web page hasn't started to load yet. When the value of counter reaches the value of counterSteps it means that timeLoaded miliseconds have passed since we started showing the tag so the web page loading can start: We set the url attribute of the webview (implemented in main.qml) to webSiteUrl so that the web view starts loading the web page. And we also give some feedback to the user by displaying the message “Going to Youtube” (for example) next to the red/green circle and in a text box on the bottom of the window (textMessage implemented in main.qml). We mustn't forget to set the value of counterIncrementing to false. Otherwise the green circle would continue growing and there would be a problem. When we get out of the if statement, we simply increment the counter.

At the end of the file we have the onTranformChanged method which is called anytime the tag is visible and there is a small change in the tag's geometric properties on the screen. Those properties are the tag's rotation, scale and translation values. Actually it is constantly called when a tag is being shown to the camera since it is impossible for a human user to keep the tag exactly in the same position. That method will draw the red and the green circles on the position where the tag is on the video output. The ChilitagsObject component has a property called transform which is a 4x4 matrix containing all the information about the tag's rotation, scale and translation. Here we extract the 3D position information from that matrix (the first three elements of the fourth row) and we convert them into 2D position on the screen by doing some approximative calculations. And finally we just call the method putCircleAtPosition from the main.qml file which will draw the red and the green circle where we want them to be on the screen.

Main.qml

This is something like the main file in this project. Even if main.cpp is the file which is run, main.cpp does nothing else than “calling” this main.qml file. At the beginning of the file we have all the necessary import statements. Then we have the body of the Window component. This Window component represents the window that will be shown when the application is run. At the beginning there are some basic attributes like the width, the height. In order to prevent some camera resolution problem, we put the maximum height to 480 and the maximum width to 640 (it is the camera's resolution). Then we have the started and pause boolean properties which are used exactly according to their names. The pause property is used in the method onVisibilityChange of ChilitagsWebsite objects. If the application is in pause then that method won't do anything. As for started, it is just used to assign the tags to the urls according to the tags.txt file at the beginning of the application. We can see the onActiveChanged method which will be called at the beginning but never again since the started property will always be false. This is probably not the best way of doing it but there is a reference problem if we use Component.onCompleted method. It is due to the FileIO class which is included dynamically as a qml component. When we write FileIO.read(), FileIO is still “null” when Component.onCompleted is executed. So when onActiveChanged is called, if the application hasn't started yet, then the tags are assigned properly with the assignTagsToUrls function defined in the same file.

Then we have the chilitagsWebsites property which is a list of ChilitagsWebsite components. The problem with that list is that it is static so we have to manually add as many ChilitagsWebsite {} objects as needed inside. But I haven't found a clear way of doing this with dynamic lists. I am not really sure dynamic lists of components exist in QML.

The Camera component is simple. It represents the camera that will be used by the application. We just indicate that its resolution is “640x480”.

Then there is the Chilitags object. That object is used as a container holding all the tags that can be detected by the camera. It has the chiliobjects property which is the list of all the tags. That list will be chilitagsWebsites.

The timer coming after Chilitags is used for the pause state of the application. It is triggered every half a second and it calls the method getPauseStateFromServer. That method will do an XML request to read the content of the pause.txt file. If the first 5 characters of that file are “pause”, then the pause property will be true and the web browser will become invisible (method activatePause). Otherwise the pause property will be set to false and the web browser will become visible. This will allow all the tags in chilitagsWebsites to behave accordingly. For example, if pause is true then they know that they shouldn't load a page when the tags are shown. Note that the XML request is done on a local file for now. The address starts with “<file:///>”. If we want to replace that and to make it work on a real http address, we have to use real url addresses.

Then there are some helper methods. PutCircleAtPosition was already mentioned. It is used by onTransformChanged in ChilitagsWebsite.qml to draw the red and the green circle on the screen. It simply sets the values redCircle.x, redCircle.y, greenCircle.x and greenCircle.y and also greenCircle.width according to the x, y and counter values given as parameters. The increaseBrowserSize and increasVideoOutputSize are quite similar. The first one is called when a tag becomes invisible and we don't need the video output anymore. It increases the browser size so that is it as wide as the window and put the camera in the upper left corner. It also activates the scroll bar of the browser. Without that we couldn't scroll on a long web page. As for increaseVideoOutputSize, it is the opposite. It is called when a tag appears in the camera view. It increases the camera size and put the upper left corner. It also deactivates the scroll bar so that we can better see if something is happening in the browser. Note the z propertiy of qml components which ensures that one view will not occlude the other when its size is increased. The smaller one will always be “on” the bigger one, its z-value will be 1. We can imagine the QML world as a 3D world where the z-axis is perpendicular to the screen and oriented towards us.

The method assignTagsToUrls is called at the beginning of the execution. It reads the content of the tags.txt file and assign the properties of the elements of chilitagsWebsites list accordingly. The split method is used to make an array of strings where every element is one line of the file. This method expects the the tags.txt file is written the following way:

tag\_21

http://www.jeuxvideo.com/

jeuxvideo.com

-------------------------

tag\_28

https://web.geogebra.org/#geometry

Geogebra

-------------------------

…

and it also expects that the number of lines in that text file is 4 \* the number of elements in chilitagsWebsites. Note that for one tag-url pair we have the following format:

tag\_ID

url

message

-------------------------

where ID should be the tag ID, url the web page url and message the “human-readable” name of the web page.

Then there is a Rectangle component which will contain the video output, the red circle and the green circle. It has the integer properties initialWidth and initialHeight which are used to avoid hard-coding when we call increaseVideoOutputSize or increaseBrowserSize. The video output has camDevice (already defined above) as its source and chilitagsConfigFile(the Chilitags Component we defined above) as its filters attribute. This is a way to tell the video output that it should do some image processing according to that Chilitags “rules”. In short if we don't set it like that, the camera won't detect the tags we show. The red circle is defined as a Rectangle component. In QML, if we set a radius property to a Rectangle it becomes a circle. And that is what we do here. We also define some basic properties like width, color, borderColor, etc. The x and y properties of that circle will be changed by the putCircleAtPosition function. Note that it also contains a Text component which is used to display some feedback messages directly at the tag's position. The green circle is similar. It has the same x and y values as the green circle but its width (and thus its radius) are manipulated by the putCircleAtPosition method which will set the width of the green circle according to the counter value (how much time the tag was shown – the longer that time is, the bigger the green circle gets).

The two last Window components are the scroll view containing the web browser and the rectangle containing the feedback messages. The scroll view, as the video output rectangle, has some initial height and initial width (by default they are 480 and 640) which are used in order to avoid hard-coding. The web browser is represented by a WebView component which has a url attribute. If we want to load a web page we simply have to set that url attribute to the address we want to load. It also contains a onLoadingChanging method which is used to write “Doing nothing” as a feedback message whenever the browser is not loading anything.

And finally, the last Window element is the rectangle displaying feedback messages. If we want to display a message in that space, all we have to do is to write 'textMessage.text = “the string we want“;'. Notice that the opacity of that text view is 0.4 so that it doesn't occlude what's happening behind on the camera or the browser. It also has a z-value which is equal to 2 so that it is always visible, whatever the state of the video output or the browser.

This project was interesting but it wasn't really what I expected. The first part of the project was about getting used to Linux and installing the necessary tools. I had to go through some step-by-step documents which were a little confusing since I had never really used Linux before. Those make and make install commands were very unclear for me. Actually, doing something without knowing exactly what I am doing can be a little discouraging. For Open CV, Chilitags and Qt, I had to clone the source code from the git repository and to install those tools on my machine. This process was very long. For example, for compiling Qt, I had to leave my computer on during the night. I also had to get used to QML which was not very difficult and it is an interesting and powerful way of creating window-based applications.

At the beginning it was very unclear what the project was about. First we talked about a project called Paprika which was about some tag functionalities developed in Javascript and the idea was to do that same thing but in Qt. In the first weeks I had to install the tools and get used to them. Running into problems was not rare. Some of them were easily solvable but others were not so easy. For example, having an error in the middle of an installation and not understanding what it is can be annoying.

After the installation, Luis told me to start programming something strange: a window with a browser and a camera which gets bigger when a tag is detected. There were some other challenges as trying to run Javascript code on the webview and the solution was to import the QtWebKit.experimental package. There were also some strange problems where I had no other choice but to go back and try something else. For example, with that browser-camera thing, I had the problem that I could use a CVCamera (Computer vision camera QML component developped by CHILI lab), I could use a web view but I couldn't use both of them at the same time. So I had to choose an other branch of qml-chilitags which did not have that bug. But still at that point the goal was not very clear.

And then came the most difficult thing: trying to run the application on Windows because it is the OS the school uses. It looked easy at the beginning but then I realized that it was not easy at all. First we tried to do that with a cross-compiler called mxe. The project was compiling but it wouldn't run on Windows, which was very strange. When we clicked on the executable, nothing happened.

At that moment we discussed about the project's goal and it started getting interesting. The idea with tessellation was not bad at all and programming the application with QML was the fun part of the project. After all I like programming and it is what I chose this project for. But there was still that Windows issue. So I programmed the application leaving that problem for the end, in a way. And finally, since the application was good on Linux, I spent the last three weeks trying to run it on Windows. I had to install Visual Studio and I ran into a lot of problems. Sometimes I was just “trying random things until it works”, which is probably not the best way to achieve something. I was able to generate a chilitags.dll file and a chilitags\_static.lib file. I tried understanding how these dll and lib files work so as to include the libraries correctly in Qt. On the end that last part didn't work but maybe I had done something wrong before that and it couldn't have worked anyway.

I learned QML which will probably be useful for my future. I was expecting to do more things in C++ but QML is cool. I realized that I should start learning some JavaScript on next holiday when I'll have some free time because that could be really helpful too. I also got some experience of what real work could be like with that running into problems pattern. But the best thing is Linux. I had Windows before and Luis installed it on my computer at the beginning of the project. Since I have it the only thing I used Windows for was this application. Otherwise I only used Linux for this project as for other courses.

You can see a log of what I've been doing for the programming and Windows part of the project on:

https://docs.google.com/spreadsheets/d/1CO4zUovPJ2W5H\_kZiLa0KEVaE0jxvBTrKYAxfGfI4N8/edit#gid=0