Implementation

Visual libraries options

There are two different options in working with images: working with a unified system that supports image processing like Mattlab, or choosing one language and then integrate image processing libraries in it.

Working with Mattlab has two problems: it is a very heavy program, that requires a lot of memory and the image processing pack is really slow. We have decided to work with an image processing library and integrate it into another language.

The image processing library used by Intel Processors is called IPL (Intel Image Processing Library). Anyone can work directly with the IPL, but it is easier to use an opensource library called Open CV which is based in the Intel Integrated Performance Primitives. Open CV is a collection of algorithms and sample code for several computer vision problems, and it will make the image processing much easier.

Open CV is a pack of DLL and some heather files. It is compatible with most of operating systems, including different versions of Linux and Windows.

We will use the Windows version, so it is easier to deploy visual applications under Windows.

Computer language options

One of the requirements of this project is that the time used to process an image has to be computationally affordable so they can use this system in the autonomous car.

Professor Rojas and a team of the Rice University Summer Institute of Statistics have tried to deploy the project in Mattlab. With this project they showed that was possible to use the Viola-Jones Algorithm to build this system, but the solution was too slow.

C++ is one language in which you have the total control of the machine. It can be programmed using basically instructions (c) or the Object orientation (c++). Object orientation is a good clearance feature, but it has the problem that it slows a lot the program results.

We deploy the system using c, and for the integration in other systems c++ so we obtain the fast feature from c and the usability of c++.

We have tried to integrate the Open CV library with two different c++ environments. First of it was Borland c++ 6.0, but the integration does not work very easily, and finally the deployment will be under Microsoft Visual C++, which is the environment recommended by the developers of Open CV library .

Programming style

Notation Method

One of the most difficult things in computer science is to modify an existing code, and it increases the difficulty if the code has been generated by other person. In the way of making this job easier we have implemented the full code using the Hungarian Notation.

Hungarian Notation takes his name from Charles Simonyi, who was born in Hungary, and it is credited as the first one to carry about this notation method. It consists in a variable naming convention that includes c++ information about the variable in its name (such as data type, whether it is a reference variable or a constant variable, etc).

Every company has his own codification method, and we have used the following:

Initial	Meaning
i	Integer
d	double
c	char
b	boolean
p	pointer

This information can be combined to make more complicate types, for example ppi means pointer pointer integer.

For long variable naming we have used capital letters at the beginning of each word. For example a pointer to an image that is going to be the grayscale of a picture we have named it pGrayscaleImage.

There are some exceptions in variable naming. The most important are the iterators in a loop (I,j,k,l ....) and the coordinates pointing to an image (X,Y,Z)

Function Definition

One important thing in the function definition is the easy treatment of the errors generated by the function. One way to implement this feature is force that all the functions that can return an error return an integer as a data type. When an error occurres, the return value of the functions is negative, and if there is no error it is positive. The return parameters of these functions have to be implemented as referenced parameters.

We have implemented this method in our project, only changing it for functions that should return a unique integer positive value. In this case when we return a positive value or a zero it means everything was right, and if we return a negative value it means an error has occurred.

For each function definition we add a little comment about what it is doing, and what possible errors it can generate. These explanations are in the heather file associated with the implementation of the function

Program constants and errors constants definitions

Constants are always difficult to treat in a project if we do not know exactly where to find them. In our thesis we use two kind of constants, one for defining program input parameters (like image size, or number of scales) and another to define the different errors that our functions can return.

In a way to simplify the modification of these parameters or understand what kind of error did a function return we have put them together and spitted them in two different heather files: ProgramParams.h and ErrorConstants.h

All the constants have been defined with the #define command of c++, in exception of the arrays that have been defined with the const modifier in the parameter type.

When we are looking in the code it is easy to identify a constant because the full name it is written in capital letters.

Module Implementation

To make maintenance project easier we have split the heathers in different files .h files and implemented all the functions in only one .cpp file.

The following list explains what there is in every module:

- ViolaJones.cpp: Implementation of all the functions needed in this project
- ViolaJones.h: Heather file associated with ViolaJones.cpp which makes including of all the files necessaries for the implementation.
- Feature.h: Definition of the type Feature. This type is necessary in order to have in memory the different features we are checking
- Cascade0.h: Implementation of the level0 cascade.
- Cascade1.h: Implementation of the level1 cascade.
- Cascade2.h: Implementation of the level2 cascade.
- Cascade3.h: Implementation of the level3 cascade.
- Cascade4.h: Implementation of the level4 cascade.
- ProgramParams.h: contains the program parameters constants as previously described
- ErrorConstants.h: contains the error constants as previously described

System Test Plan:

Basic functions test plan:

The final step in all the projects is to test how it works with real live images. During the time we spend deploying this project we have also taken some pictures in the street. As the Urban Challenge is going to be in Berlin, all the pictures are from this city.

The testing process used consists in testing each of the functionality alone, and then make an incremental test adding new functionalities each time. The steps we have done are the following ones:

- Image Acquisition Test: We have tested that we can open different format types. We return an error if the image is not colored or it has not the specified sizes.
- Grayscale Generation: We have tested that we can convert colored images to Grayscale images.
- Binary red image generation: We have tested that the red image returned is a really binary red image, only with the red pixels. Each image generated has been manually compared with its original one
- Integral image generation: To test this functionality we have changed the code a little bit. As we need to check the image visually we have generated an integral image where the maximal pixel value is 255. By this way, we have generated some images with a few grey points, and then we have checked that the integral image is good generated.

  After this first step, we have ensured that with light pictures the functionality works. To test it with real images we have changed back previous modifications, and tested it with real pictures. The only way to ensure that an integral image is being good generated is to save the integral image in a CSV file, and manually check it using Microsoft Excel.

For checking each of these features we deploy a visual application that allows us to call every function. We can see the human interface used in Image 22.

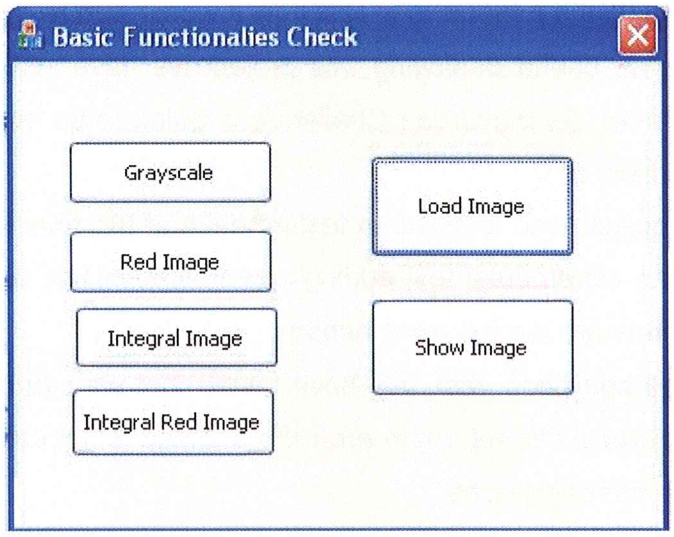

Image 22: Basic functionality testing system

Viola Jones function testing plan

Once we have tested all the basic operations we have to test the main function of the system. We have tested it by two different ways. The first one consists in testing each feature alone. By this way, we can check that all the feature representations have been good implemented, and each of it returns to a group of possible car locations. This test has been done using all the possible scales and all the possible sub windows, ignoring the simplification process described previously. After we have checked that every single feature works with all the scales, we check them together to find the real position of the sub windows.

The final test procedure we do is to convert the standard window size returned by the Viola Jones Algorithm, into a more accurate size, applying the size adjusting described previously in this paper.

Results

The test plan has been run under the following conditions:

Camera**:** Casio EXILIM with a resolution of 7.2 Mega pixels. This resolution generates images of 3072x2304 pixels

Computer: Sony VAIO serie VGN-FE running windows XP sp2 Spanish version

The pictures have been taken between January 08 and March 08 in the city of Berlin.

Car recognizing results:

Number of Car images: 133

Number of Cars in these images: 154

Number of Cars recognized in these images 149

Recognizing percentage: 96%

Number of Non-car images: 140

Number of Cars in these images: 0

Number of Cars recognized in these images: 1

Error recognizing percentage: 0,007%

Computation time results:

Average of Non-car images spend time: 8s

Average of car images spend time: 7,3s

Conclusions:

We have accomplished all the goals from this project.

We have developed an artificial vision system based in “Robust Real-time Object Detection” project, developed by Viola and Jones, using c++ with OpenCV library, which allows us to detect cars in a computable affordable time.

As it was specified for each recognized car we give its position and its size. This allows us to know how far from the camera the other cars are.

We have contributed with this project to do a better adjusting of the car’s size, using the Canny algorithm after locating the car and doing a projection analysis.

In conclusion, we can say all the goals for this thesis have been accomplished