Edge detection

Authors : Ophélie Thierry

Introduction

Several general notions about the edge detection

Image processing is one of the most important fields in the domain of computer vision[1]. Most scientific domains use information extracted from images in one way or another. For a computer to make sense of these images, and be able to extract meaningful data from them, it needs to be able to interprete and understand them. That is where Image Processing comes in, allowing a computer to process an image and detect its major features and reducing mistakes, variance linked to the experimenter leading to less biased conclusions, perform higher-level vision tasks like face recognition [2] [3]. In our project, we will examine one specific field of image processing called edge detection.

The physical notion of edge comes from the shape of three dimensional objects or from their material properties. But, seeing as the acquisition process translates 3D scenes to 2D representations, this definition does not apply to image processing. In this report we will use the following definition by Bovik (2009): "An edge can generally be defined as a boundary or contour that separates adjacent image regions having relatively distinct characteristics according to some features of interest. Most often this feature is gray level or luminance"[4]. According to this definition, the pixels of an image belonging to an edge are the pixels located in regions of abrupt gray level changes. Moreover, to avoid counting noise pixels as edges, the pixels have to be part of a contour-like structure. Edge detection is the process of finding the pixels belonging to the edges in an image, and producing a binary image showing the locations of the edge pixels.

Among all the implemented image analysis software, ImageJ is one of the most used as one of the most ancient, free, open-source, easy to use and with an extansive architecture implemented in Java. It's a generalist software but the differents plugins coded by the community allow it to take care of a wide range of images and be enough configurable to become very efficient and specific [5] [6] [7] [8].

[1] Bovik AC, editor. The essential guide to image processing. Academic Press; 2009 Jul 8.

[2] Doherty G, Mettrick K, Grainge I, Lewis PJ. Harwood C, Wipat A (Ed). Chapter 4 - Imaging fluorescent protein fusions in live bacteria. In Methods in Microbiology, Systems Biology of Bacteria. Academic Press. 2012; 39(4):107-126

[3] Schindelin J, Rueden CT, Hiner MC, Eliceiri KW. The ImageJ ecosystem: An open platform for biomedical image analysis. Molecular reproduction and development. 2015 Jul 1;82(7-8):518-29.

[4] Bovik AC, editor. The essential guide to image processing. Academic Press; 2009 Jul 8.

[5] Schneider CA, Rasband WS, Eliceiri KW. NIH Image to ImageJ: 25 years of Image Analysis. Nature methods. 2012;9(7):671-675.

[6] Eliceiri KW, Berthold MR, Goldberg IG, et al. Biological Imaging Software Tools. Nature methods. 2012;9(7):697-710. doi:10.1038/nmeth.2084.

[7] Rueden CT, Schindelin J, Hiner MC, DeZonia BE, Walter AE, Arena ET, Eliceiri KW. ImageJ2: ImageJ for the next generation of scientific image data. BMC Bioinformatics. 2017; 18:529.

[8] Kainz P, Mayrhofer-Reinhartshuber M, Ahammer H. IQM: An Extensible and Portable Open Source Application for Image and Signal Analysis in Java. Martens L, ed. PLoS ONE. 2015;10(1):e0116329. doi:10.1371/journal.pone.0116329.

Interest of changing the implementation language.

Currently, ImageJ is mainly present as a desktop software downloadable. However, there is an applet which can be used by Java-enabled browser [9]. Among the different implementation langage, Java and Javascript share only several expressions in their langage, but Java is a compiled langage and less stringent on its typing and doesn't have a portability as good as the JavaScript which is an interpreted langage and a major component in the development of curent application running within a web browser [10]. Implement the ImageJ functions in JavaScript could improve the efficiency of this applications and reduce their calculation time [11]. In the Edge Detection case, it was demonstrated in the second repport.

A supplemental way to improve the efficiency of this applications is to use WebGL, an API allowing to execute scripts in JavaScript on the Graphic Processing Unit (GPU), allowing the parallelization of the operations executed, reducing the calculation time and improve rendering performance and quality [12] [13]. Also WebGL is well supported by the main web browser [14]. Various ameliorations were add in the WebGL2 version as the multiline template literals, the standardisation of the Vertex Array Objects or possibility to create Uniform Buffer Objects.

Aim of the project

In our project, we began by documenting the main linear edge detection approaches and algorithms, and their implementation in the image processing software ImageJ [15]: convolution with edge templates (Prewitt [16], Sobel [17], Kirsch [18], Robert's Cross [ˆROB1963]), zero-crossings of Laplacian of Gaussian convolution[19], zero-crossings of directional derivatives of smoothed images (Canny[20]).

We then performed a benchmark on the ImageJ plugins, in order to compare their execution time and the memory load for the Java Virtual Machine (JVM). For the second and third part of our project, we made respectively our own implementations of the Sobel, Prewitt, Robert's cros, Canny and Laplacian of Gaussian

[9]Schneider CA, Rasband WS, Eliceiri KW. NIH Image to ImageJ: 25 years of Image Analysis. Nature methods. 2012;9(7):671-675.

[10]Bienfait B, Ertl P. JSME: a free molecule editor in JavaScript. Journal of Cheminformatics. 2013;5:24. doi:10.1186/1758-2946-5-24.

[11]Wollny G, Kellman P, Ledesma-Carbayo MJ, Skinner MM, Hublin JJ, Hierl T. MIA - A free and open source software for gray scale medical image analysis. Source Code Biol Med. 2013; 8:20. doi:10.1186/1751-0473-8-20.

[12]Shamonin DP, Bron EE, Lelieveldt BPF, et al. Fast parallel image registration on CPU and GPU for diagnostic classification of Alzheimer's disease. Frontiers in Neuroinformatics. 2013;7:50. doi:10.3389/fninf.2013.00050.

[13]Rose AS, Hildebrand PW. NGL Viewer: a web application for molecular visualization. Nucleic Acids Research. 2015;43(Web Server issue):W576-W579. doi:10.1093/nar/gkv402.

[14]Shi M, Gao J, Zhang MQ. Web3DMol: interactive protein structure visualization based on WebGL. Nucleic Acids Research. 2017;45(Web Server issue):W523-W527. doi:10.1093/nar/gkx383.

[15]Schindelin J, Rueden CT, Hiner MC, Eliceiri KW. The ImageJ ecosystem: An open platform for biomedical image analysis. Molecular reproduction and development. 2015 Jul 1;82(7-8):518-29.

[16]Prewitt JM. Object enhancement and extraction. Picture processing and Psychopictorics. 1970 Jan 1;10(1):15-9.

[17]Sobel I. An isotropic 3× 3 image gradient operator, presentation at Stanford Artificial Intelligence Project (SAIL).

[18]Kirsch RA. Computer determination of the constituent structure of biological images. Computers and biomedical research. 1971 Jun 1;4(3):315-28.

[19]Marr D, Hildreth E. Theory of edge detection. Proceedings of the Royal Society of London B: Biological Sciences. 1980 Feb 29;207(1167):187-217.

[20]Canny J. A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence. 1986 Nov(6):679-98.

operators using ECMAscript6[21] and WebGL2 [ˆHAL2014].

This repport will be focused on the three convolutions with edge templates : Prewitt, Sobel and Robert's Cross. Informations about the other edge detection algorithms can be seen in the previous repports.

The link to our github repository containing our reports in markdown format, the images, and the code for the benchmark and algorithms is : https://github.com/bockp/Edge-Detection-project.

Material & Methods

Implementation of the functions

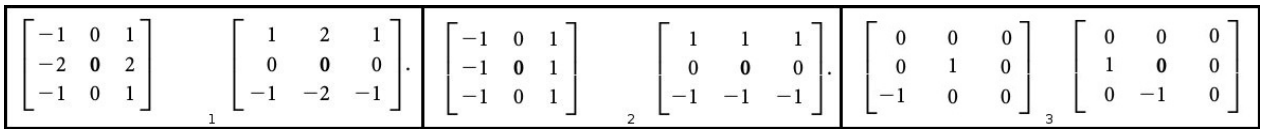

Fig.1: Horizontal and vertical kernels 2D kernels : 1:Sobel operator, 2:Prewitt operator, 3:Robert's cross operator

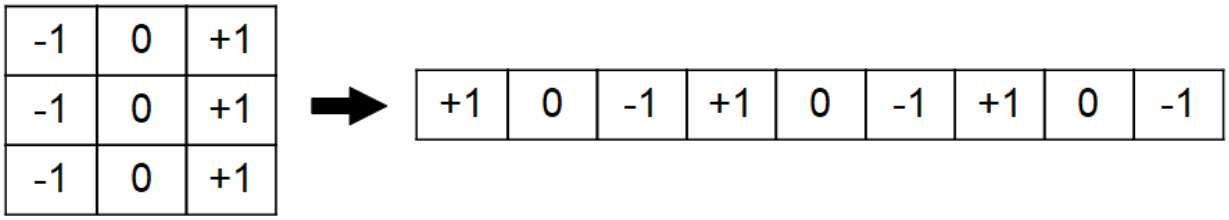

Fig.2: Implementation of the Prewitt horizontal kernel in a list.

The corresponding script is in the */src/gpu/edge.js* file, and the related exemples in the *exemples/* repository in the files *testEdge.html* and *testEdge.js*.

As the three convolution share the same process, a general function has been implementated, taking into argument the raster, the graphical environment and a specific kernel. This function will be called by three functions, each one specialized in a specific convolution. All the specific vertical and horizontal kernels for each convolution are implemented as lists in constant global variable outside the main functions. In order to facilitate the implementation, the six lists have the same size, and zero are add to the Robert cross kernels. The result is an image in which the edges have high pixel values compared to the rest of the image.

In order to facilitate the localization of the result in the webpage, the gpuEnv where the picture has to be displayed is takken as argument for each function.

gpuEdgeSobel(), gpuEdgePrewitt (), gpuEdgeRobert ()

These three functions are build in the same way. Each of them take into account the image and the graphical environment, and call the *gpuEdge()* function with the lists corresponding to the specific kernel.

[21]ECMAScript EC. European Computer Manufacturers Association and others. ECMAScript language specification. 2011.

gpuEdge()

It's the general function which will be called by the three others. It take four arguments : the picture, the graphical environment where the picture has to be displayed and the vertical and horizontal kernel (*kernelV* and *kernelH*) used for treating the picture. This function use only one shader.

The vertex shader is defined as the constant *src_vs* which transform the image coordinates in order to make them compatible with the displaying in the canvas. The fragment shader, defined as the constant *src_fs*, for each coordinate get the values of the corresponding neighboring elements and use it to realise the convolution according to the values of the kernel used as argument. The final value is stored in the vec4 *outputColor* for each coordinate.

The pictures used for the benchmark are the *"coins"* pictures in uint8 with five different sizes in pixels : 128x105, 300x246, 512x420, 1024x840, and 2048x1679. The benchmark process was encoded by Cecilia Ostertag.

Results

Fig.3: Result of the implementation of WebGL edge detection process (respectively from the left to the right : without processing, with prewitt, sobel and robert operators) on uint8 pictures.

Fig.4: Result of the implementation of WebGL edge detection process (respectively from the left to the right : without processing, with prewitt, sobel and robert operators) on float32 pictures.

The figures [Fig.3] and [Fig.4] show the result of the process on uint8 and float32 pictures. When the process is used on uint16 pictures, the process report an error and refuse to display the result. For each format, the prewitt filter give the brightest result.

EdgeDetectionMethod	128px	300px	512px	1024px	2048px
Prewit (WGL2)	3.6	25.6	83.3	261.3	1193.7
Sobel (WGL2)	3.7	24	75.7	252.5	1268.2
Robet (WGL2)	5.3	23.4	75.7	249.8	1097.3
Prewit (JS)	50.3	50.8	51.2	65	204.4
Sobel (JS)	52	49.5	55.5	73.4	187.6
Robet (JS)	43.3	39.4	44.3	60.4	118.1

Tab.1: Average the different execution times in ms of the differents implemented process in Javascript (JS) and WebGL2 (WGL2). Ten repetitions have been done on pictures of different sizes.

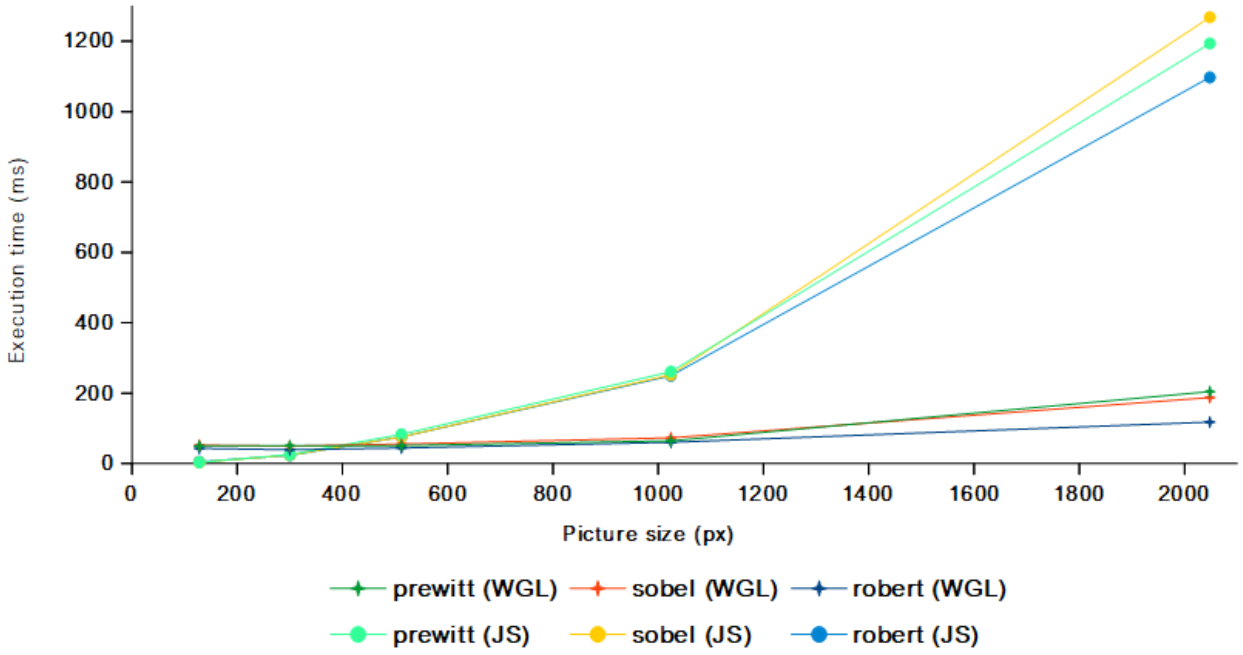

Fig.3: Comparaison of the execution time (in ms) of the implemented process in Javascript (JS) and WebGL2 (WGL2) according to the size of the picture (px).

Without taking into account the 2048x1679px picture, the correlation coefficient calculated remain superior to 0.97 with a slope of nearly 0.30 for each implementation in JavaScript, whereas respectively 0.80 and 0.02 for the WebGL implementation.

Discussion and Conclusion.

For pictures of small size (less than 500x500px), the implementation in Javascript seem to consume less calculation time than the WebGL2. But the calculation time increase rapidly for the process implemented in JavaScript according to the image size, unlike for the implementation in WebGL2 for which the calculation time remain generally constant.

For small size pictures, the calcul time consumption seem correlate to the picture size for the JavaScript implementation (until 1000x1000px at least), unlike the WebGL2 implementation. It will be interasting to assess the correlation of the calculation time with the size of the picture, increase the number of repetitions used for the benchmark and use a wider range of small picture in order to improve the correlation calculation for the curve.

For each implementation language, the Robert cross seem to be the most advantageous in calcul time consumption, but not according to the display, where the orther seems to highlight more details than it.

Several improvement are still needed in order to make the process works on uint16 pictures.

References

Used in the present repport

Used in previous repports