**MTI805 – Compréhension de l’image**

**Laboratoire 1**

**Détection de primitives simples et faciale**

|  |  |
| --- | --- |
| **Session** | **Hiver 2017** |
| **Groupe** | **01** |
| **Numéro du travail** | **Laboratoire 1** |
| **Étudiants** | **Etienne Dewaele**  **Tony Duong** |
| **Courriel des étudiants** | **etienne.dewaele.1@ens.etsmtl.ca**  **tony.duong.1@ens.etsmtl.ca** |
| **Chargé de cours** | **Luc Duong** |
| **luc.duong@etsmtl.ca** |

# Introduction

# Description de l’approche

## Détection de lignes

Pour effectuer la détection de primitives simples comme les lignes et de cercles dans une image, nous avons utilisé la transformée de Hough qui gère très bien cette tâche. Cette technique, bien qu’étant assez vieille (inventée en 1962 par Paul Hough), reste une référence dans le domaine de la vision par ordinateur. C’est la raison pour laquelle nous avons choisi d’étudier et d’appliquer la transformée de Hough pour la détection de lignes et de cercles dans notre laboratoire.

Avant de plonger dans le programme, étudions d’abord de quelle façon les lignes sont détectés dans une image par application de cette méthode.

### Transformée de Hough

En vision artificielle, l’une des premières problématiques a été de détecter de simples primitives comme des lignes, des cercles ou encore des ellipses dans une image. En effet, la détection de telles primitives est parfois à la base pour la détection d’objets plus complexes et également pour la reconnaissance de formes et d’objets. La transformée de Hough est une technique datant de 1962 qui a été inventée pour répondre à ce problème. Au départ, celle-ci a été conçue pour la détection de lignes mais des extensions ont été apportés quelques années plus tard pour détecter d’autres formes géométriques comme des cercles, des ellipses.

### Théorie

Avant d’appliquer la transformée de Hough, il est préférable pour des soucis de performance d’appliquer un prétraitement de détection d’arêtes comme le filtre de Canny. Cela faciliterait la détection de lignes car seulement les contours des formes et objets seront conservés.

En mathématiques, une droite dans un plan peut être exprimée soit dans le système cartésien soit dans le système polaire. Etant donné que représenter une ligne parfaitement verticale dans le système cartésien, la transformée de Hough exprime donc les droites (ou lignes) dans le système polaire. Une équation d’une droite peut être écrite comme :

est la distance de la droite par rapport à l’origine.  
 est l’angle en radians de la droite passant par l’origine et le point le plus proche de la droite et l’axe des abscisses.

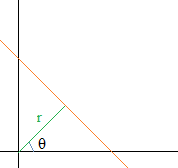


Figure – r et θ illustrés

Tous les points des contours (après le filtre de Canny) seront projetés dans un autre plan des coordonnées polaires (r, θ). Leur projeté sur ce nouveau plan représente toutes les droites possibles passant par ce point et est représenté par une courbe d’équation :

Où varie entre à , et et sont les coordonnées du point traité ( et sont fixes).

|  |  |
| --- | --- |
| Angle | Distance |
| 0 | r1 |
| 45 | r2 |
| 90 | r3 |
| 135 | r4 |

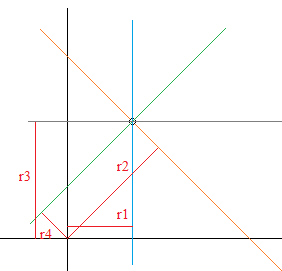


Figure - Echantillon de toutes les droites possibles passant par un point

Ainsi, avec cette équation, les courbes pour chaque point de l’image originale (celle après application du filtre de Canny) seront tracées dans le plan transformé. Dans la figure 3, on suppose qu’il n’y a seulement trois pixels dans l’image et que ceux-ci sont alignés.

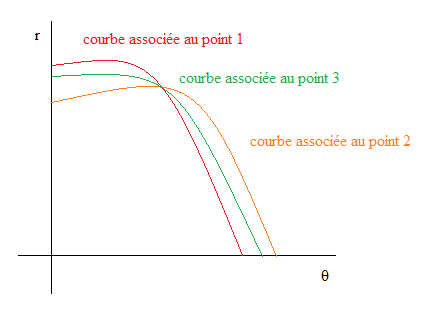


Figure – Plan transformé

Une fois toutes les courbes tracées, la dernière étape consiste à trouver les points d’intersection qui représenterait une ligne passant par les points associés aux courbes qui s’intersectent. En effet, si plusieurs courbes s’intersectent, cela signifierait que leurs paramètres r et θ seraient identiques et par conséquent qu’une droite d’équation passerait par tous les points (associés à ces courbes. Cette droite est une ligne dans l’image originale.

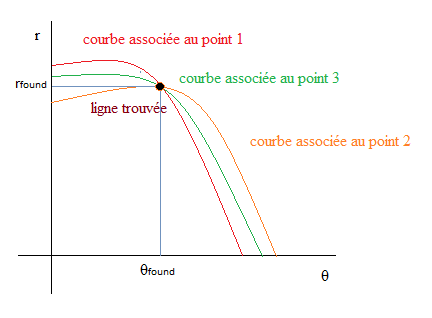


Figure - Intersection de plusieurs courbes

En général, dans une image quelconque, il est quasiment certain de trouver plusieurs pixels alignés à échelle minuscule même si à échelle normale, ces pixels ne forment pas de lignes. Pour pallier à cela, il suffit de déterminer un seuil qui serait le nombre minimum d’intersections pour confirmer que ces pixels forment bien une ligne.

### Programme OpenCV

Une fonction void detectLines(cv::Mat& frame) a été créée pour détecter toutes les lignes dans une frame.

1. La première étape est de lisser l’image afin d’éliminer le bruit. La présence de bruit pourrait perturber le fonctionnement de la détection de contours qui se trouve à l’étape suivant et les contours ne seraient pas clairement détectés.

Pour effectuer cette tâche, la fonction OpenCV

void blur(  
InputArray src,   
OutputArray dst,   
Size ksize,   
Point anchor=Point(-1,-1),   
int borderType=BORDER\_DEFAULT )

fait très bien l’affaire.

1. Ensuite, la méthode de Canny est appliquée sur l’image lissée afin d’extraire les contours. Ceci a pour but de faciliter la détection de lignes.

Dans OpenCV, la fonction est comme suit :

void Canny(  
InputArray image,   
OutputArray edges,   
double threshold1,   
double threshold2,   
int apertureSize=3,   
bool L2gradient=false)

1. Finalement, la méthode de Hough est appliquée sur l’image résultante de la fonction Canny().

void HoughLines(  
InputArray **image**,   
OutputArray **lines**,   
double **rho**,   
double **theta**,   
int **threshold**,   
double **srn**=0,   
double **stn**=0)

lines contient un tableau à deux dimensions composé des paramètres r et θ comme présenté dans la partie théorique.

1. Il reste maintenant à tracer les lignes via la méthode

void line(Mat& **img**, Point **pt1**, Point **pt2**, const Scalar& **color**, int **thickness**=1, int **lineType**=8, int **shift**=0)

### Résultats et discussion

Le cas suivant montre l’importance de bien ajuster les paramètres

threshold1 threshold2 de Canny()

qui correspondent respectivement aux seuils inférieur et supérieur. Lorsque les deux seuils sont fixés à 0 (par défaut), tous les pixels quasiment sont acceptés comme « pixel de contour » (Figure 5). Cela résulte à un résultat complètement faussé de la détection des lignes. En effet, beaucoup de points « parasites » sont alignés, ce qui donnerait l’illusion au système que ces points sont alignés.

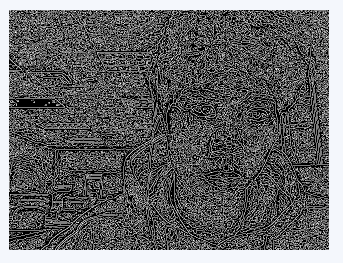


Figure - Image résultante de Canny() - sans ajustement des seuils



Figure - Lignes détectées

Ainsi, après avoir ajusté les paramètres jusqu’à avoir un résultat correct, on obtient un résultat beaucoup plus satisfaisant. Les paramètres choisis sont :

threshold1 : 70

threshold2 : 250

Les contours sont détectés même s’il persiste quelques contours non détectés (lignes entre le mur et le plafond). Les pixels ont par contre été toutes supprimés. Il suit donc une détection correcte des lignes.

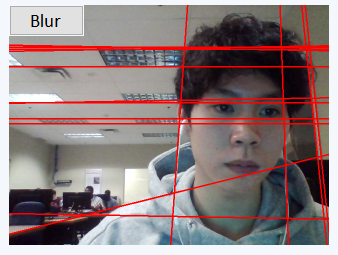


Figure - Lignes détectées



Figure - Image résultante de Canny() - avec ajustement des seuils

## Détection de cercles

Pour la détection de cercle, une dérivation de la méthode de la transformée de Hough est utilisée. La différence étant que les paramètres (r, θ) seront remplacés par les paramètres (r, a, b) où (a, b) sont les coordonnées du centre du cercle. Tous les cercles possibles intersectant un point seront donc projetés dans un espace 3D sous la forme d’un cône (analogue aux courbes projetés dans un plan 2D).

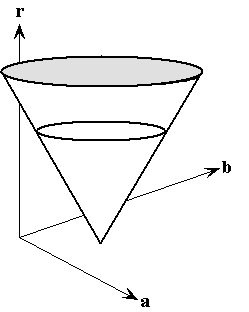


Figure - Espace transformée dans le cas de la détection de cercles

### Programme OpenCV

Une fonction void detectCircles(cv::Mat& frame) a été créée pour détecter des lignes dans une frame.

1. Comme pour la détection de ligne, il est nécessaire de lisser l’image en premier lieu afin de supprimer le bruit pour les mêmes raisons. Pour effectuer cette tâche, la même fonction OpenCV est appelée.

void blur(InputArray src, OutputArray dst, Size ksize, Point anchor=Point(-1,-1), int borderType=BORDER\_DEFAULT )

1. Contrairement à l’étape précédente, la méthode de Hough pour la détection de cercle applique déjà à l’interne la fonction Canny() pour détecter les contours. Il n’y a donc pas besoin de l’appeler explicitement et nous pouvons directement procéder à la fonction suivante.

void HoughCircles(  
InputArray **image**,   
OutputArray **circles**,   
int **method**,   
double **dp**,   
double **minDist**,   
double **param1**=100,   
double **param2**=100,   
int **minRadius**=0,   
int **maxRadius**=0 )

Cette fois ci, circles contient un tableau à trois dimensions composé des paramètres a, b et r qui représente les coordonnées du centre des cercles ainsi que la valeur de leurs rayons.

1. Il reste maintenant à tracer les cercles via la méthode

void circle(Mat& **img**, Point **center**, int **radius**, const Scalar& **color**, int **thickness**=1, int **lineType**=8, int **shift**=0)

### Résultats et discussion

Comme pour la détection lignes, la fonction HoughCircles() nécessite quelques ajustements au niveau des paramètres. Les paramètres importants à ajuster sont les suivants.

double **minDist** // distance minimale entre les centres des cercles détectés

double **param1** // seuil supérieur de la fonction de Canny() qui est intégré, le seuil inférieur est la moitié du seuil supérieur

double **param2** // seuil de l’accumulateur pour la détection de cercles. Plus elle est basse, plus de faux cercles pourraient être détectés.

int **minRadius // rayon minimum des cercles à détecter**  
int **maxRadius // rayon maximum des cercles à détecter**

Nous allons nous concentrer sur param2 en particulier. Le résultat ci-dessous est obtenu quand celle-ci est fixé à une valeur très basse (15 dans ce cas). Les autres paramètres sont fixés à des valeurs raisonnables.

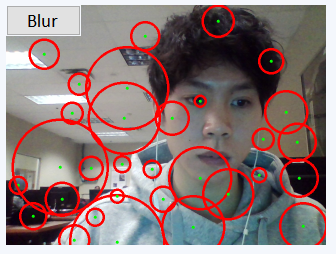
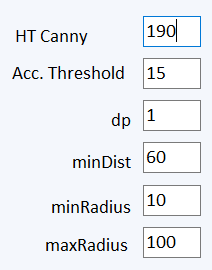


Figure - Paramètres

Figure - Faux cercles détectés

Maintenant, augmentons le seuil de l’accumulateur à 60 pour rendre plus difficile la détection et ainsi rejeter les faux positifs. Nous pouvons remarquer une nette amélioration. Puis quand un réel cercle apparait, nous pouvons remarquer qu’il le détecte bel et bien. Nous avons donc constaté la manière d’utiliser ce paramètre.

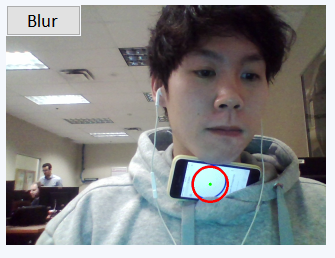
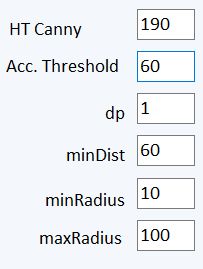


Figure - Seuil accumulateur ajusté

Figure - Bonne détection



Figure - Suppression des faux cercles

## Détection de visages

Le sujet de la détection faciale est encore en développement de nos jours. En effet, même si les premières expérimentations datent des années 1970, il reste encore beaucoup de marge de progression dans ce champ d’application. En 2001, Paul Viola et Michael Jones ont franchi un nouveau cap en publiant leur méthode basée sur un entrainement long et complexe du système. Cette méthode reste encore à ce jour la base de la détection d’objets et de formes bien qu’étant originellement conçu pour la détection de visages. Beaucoup d’améliorations de celle-ci ont été proposées depuis. Dans notre application de ce laboratoire, cette technique a donc été naturellement sélectionnée pour la détection de visages.

Etudions d’abord la théorie qui se cache derrière cet algorithme.

### Viola Jones

Considérons un système qui n’a aucune connaissance préalable sur la structure d’un visage. La première étape est d’entrainer ce système en lui « montrant » des images de visage et des images sans visage, tout en lui indiquant si l’image montrée est un visage ou non. Le système sera arrivé à un point où il pourra discerner un visage dans une nouvelle image à partir des connaissances acquises pendant son entrainement. Ceci est une manière simple d’expliquer comment fonctionne la méthode de Viola-Jones pour la détection de visage.

#### Caractéristiques de Haar (Haar features en anglais)

Comment un tel système décide qu’une image est un visage ou non, du point de vue mathématique ? Viola-Jones ont utilisé dans leur algorithme les *Haar features* (Figure 1)qui sont des « caractéristiques utilisées en vision par ordinateur pour la détection d’objet dans des images numériques ».



Figure - Exemple de Haar features  
Par Indif — Travail personnel, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=11681210>

Elles sont appliquées dans une image et la valeur d’une caractéristique appliquée sera la somme des pixels dans le(s) rectangle(s) noir(s) soustraite de la somme des pixels dans le(s) rectangle(s) blanc(s). SHADEDRECTANGLES - CLEARRECTANGLES

#### Exemple

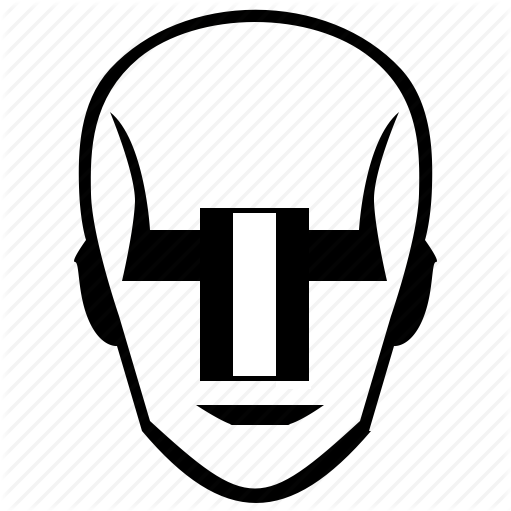
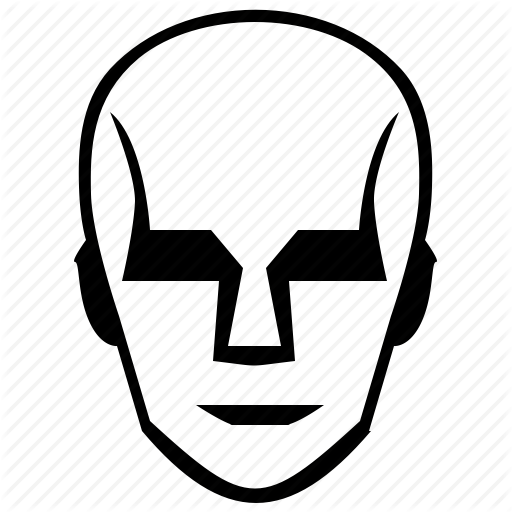


Figure – Application des caractéristiques de Haar  
https://www.iconfinder.com/icons/1162581/comics\_face\_head\_robot\_icon

La région des yeux du visage étant en général plus sombre que la région se situant juste en dessous, une caractéristique pertinente serait un *Haar feature* à deux rectangles, noir pour le dessus et blanc pour le dessous (Figure). Pour l’arête du nez, le même principe s’applique avec un *Haar feature* à trois rectangles, noir sur les côtés et blanc au milieu (Figure). Cela résulte sur une valeur haute de ces deux caractéristiques à ces endroits du visage (yeux et nez respectivement).

Ces caractéristiques par exemple, seront donc capables lors de la phase de déterminer la position des yeux et de l’arête du nez respectivement dans une image.

Les caractéristiques utilisées (Figure 1) peuvent donc très bien, lorsque modifiées en taille et couleur, « s’associer » à des caractéristiques d’un visage humain comme nous l’avons vu dans l’exemple précédent.

Elles peuvent être modifiés en dimension (taille des régions noires et blanches) et les couleurs des régions peuvent aussi être inter-changées (Figure 2).



Figure - Application des Haar features - http://comp3204.ecs.soton.ac.uk/cw/haar.jpg

L’algorithme de Viola-Jones utilise une fenêtre de 24\*24 pixels qui sera glissée à travers l’intégralité de l’image d’entrée (Figure). De plus, si tous les paramètres d’échelle, de types (couleurs) et de position sont considérés, il y aurait 162 336 caractéristiques possibles pour une seule fenêtre de 24\*24 pixels. Nous pouvons en déduire qu’il serait trop coûteux en temps de tous les évaluer lors de la détection dans une image, surtout dans le cas d’une application en temps réel comme la détection dans un flux vidéo.

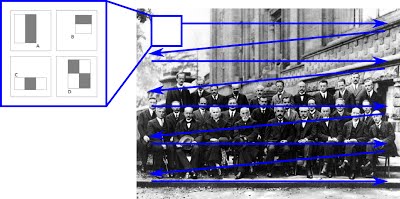


Figure – Le glissement de la fenêtre de détection illustré  
https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection#TOC-Image-Pyramid

Ceci nous amène à des méthodes d’économie de calcul.

#### Image intégrale (integral image en anglais)

Nous avons vu précédemment que les sommes des pixels dans les régions des caractéristiques doivent être calculées. Au lieu de sommer chaque pixel un à un, Paul Viola et Michael Jones ont employé la technique de l’image intégrale.

En effet, dans une image intégrale, la valeur d’un pixel à la positon (x, y) est la somme des pixels de toute la zone supérieure et à gauche de celui-ci.

## Exemple

|  |  |  |
| --- | --- | --- |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

|  |  |  |
| --- | --- | --- |
| 1 | 1+1 | 1+1+1 |
| 1+1 | 1+1+1+1 | 1+1+1+1+1+1 |
| 1+1+1 | 1+1+1+1+1+1 | 1+1+1+1+1+1+1+1+1+1 |

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 2 | 4 | 6 |
| 3 | 6 | 9 |

A partir d’une image intégrale, il est possible d’obtenir la somme des valeurs des pixels d’une région rectangulaire quelconque dans une image très rapidement à partir des valeurs des coins de ce rectangle.

C

A

D

B

A, B, C et D sont les valeurs dans l’image intégrale à leur position.

Dans l’exemple ci-dessus, pour calculer la somme de tous les pixels de la zone orange, il suffit seulement d’appliquer le calcul suivant.

#### Adaboost

Comme dit précédemment, un total de plus de 160 000 caractéristiques peuvent être généré sur une petite fenêtre de 24\*24 pixels. Le traitement d’un tel ensemble serait trop coûteux. L’objectif de Adaboost est de réduire le nombre de *features* et de ne garder que celles qui sont pertinentes.

Toutes les caractéristiques conservées sont appelées des classificateurs faibles (*weak classifiers* en anglais) et sont chacune associées à un poids.

Ainsi, la combinaison linéaire de ces classificateurs faibles multipliés chacun par leur poids associé constitue un classificateur fort qui sera utilisé pour la détection.

* est un classificateur fort.
* est un classificateur faible.
* est un poids.

Le détail de l’algorithme ne sera pas présenté dans ce rapport. Il peut être trouvé dans l’article de Paul Viola et Michael Jones.

Le traitement de cet algorithme est très long. A titre indicatif, l’entrainement du système a requis plusieurs semaines sur le système original et sur les systèmes plus modernes, cet entrainement peut aller de plusieurs heures à quelques jours.

source :https://tspace.library.utoronto.ca/bitstream/1807/33294/3/Lo\_Charles\_K\_201211\_MASc\_thesis.pdf

#### Cascade de classificateurs

Dans une image, il est évident qu’une large partie de celle-ci ne contiendra pas de visage (ou d’objet que l’on souhaite détecter). Par souci d’efficacité, l’objectif de la cascade de classificateurs est de pouvoir rejeter les fenêtres à évaluer ne contenant pas de visage le plus rapidement possible.

Si on applique un seul et unique classificateur fort composé de toutes les classificateurs faibles sur l’ensemble des fenêtres à évaluer, cela prendrait beaucoup de temps de calcul pour chacune de ces fenêtres même s’il y a absence de visage.

La structure en cascade permet de contourner ce problème.

Les classificateurs faibles sont toutes réparties dans plusieurs classificateurs forts. Les classificateurs forts sont ensuite appliqués séquentiellement sur la fenêtre de détection. Suivant l’application de chacun de ceux-ci, deux décisions sont alors possibles :

- la fenêtre contient un visage d’après le classificateur, on passe au classificateur suivant.  
- la fenêtre ne contient pas de visage, la fenêtre est rejetée et on glisse la fenêtre de détection sur l’image. On recommence le processus depuis le début.

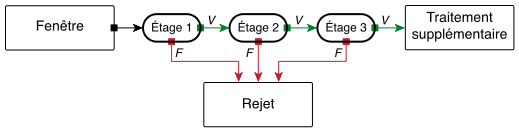


Figure - Par Indif — Travail personnel, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=11691297>

Lorsqu’une fenêtre a validé toutes les étages, elle est considérée comme contenant un visage.

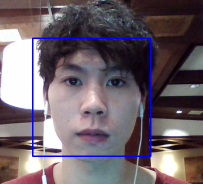


Figure - Fenêtre contenant un visage

### Programme OpenCV

Une fonction void detectFaces(cv::Mat& frame, CascadeClassifier cc) a été créée pour détecter des lignes dans une frame.

1. CascadeClassifier est un objet contenant toutes les informations de la cascade de classifieurs. En effet, OpenCV contient déjà un fichier xml appelé "haarcascade\_frontalface\_default.xml" qui est le résultat de l’entrainement. Il faut la charger comme suit.

CascadeClassifier face\_cascade;

string face\_cascade\_location = "haarcascade\_frontalface\_default.xml";

face\_cascade.load(face\_cascade\_location);

1. Dans notre programme, il n’y a donc pas besoin de traiter cette l’entrainement. Elle a déjà été faite au préalable et tout ce que l’on a à faire est d’appeler la fonction

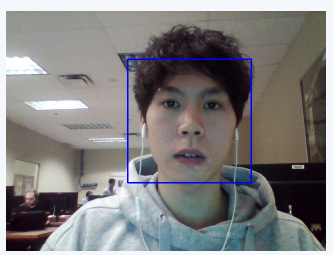
void CascadeClassifier::detectMultiScale(  
const Mat& **image**,   
vector<Rect>& **objects**,   
double **scaleFactor**=1.1,   
int **minNeighbors**=3,   
int **flags**=0,   
Size **minSize**=Size(),   
Size **maxSize**=Size())

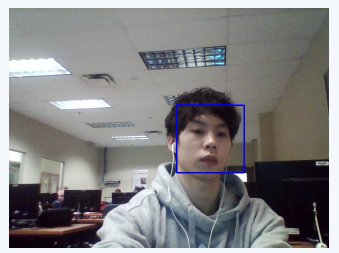
1. Il reste maintenant à encadrer les visages détectés par les rectangles via la méthode

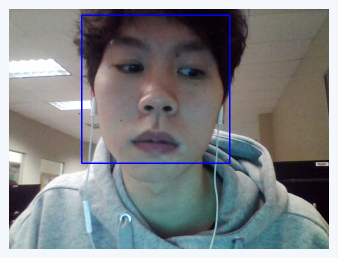
void rectangle(  
Mat& **img**,   
Point **pt1**,   
Point **pt2**,   
const Scalar& **color**,   
int **thickness**=1,   
int **lineType**=8,   
int **shift**=0)

### Résultats et discussion

Le cascade de classifieur utilisé dans notre application a été entrainé spécialement pour la détection de visage frontale. Nous allons tester les limites de celui-ci dans plusieurs situations.







Comme nous pouvons le remarquer, lorsque le visage est trop incliné de côté, en bas, en haut ou est de profil, le système a du mal à détecter celui-ci. Néanmoins, celui-ci est capable de détecter le visage efficacement indépendamment de la distance du visage à la caméra. La raison est que la taille de l’image originale est diminué à chaque fois ( nouvelle\_taille = ancienne\_taille / scaleFactor ) que la fenêtre de détection a fini son parcours de détection sur toute l’image. La fenêtre de détection elle en revanche ne change pas de taille (24\*24 pixels constamment). Ce procédé permet de repérer des visages de toute taille dans une image.

# Conclusion

# Bibliographie

Viola-Jones

[1] [P.Viola, M.Jones, Rapid Object Detection using a Boosted Cascade of Simple Features](https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf)

[2] [Wikipedia, Viola Jones Object Detection Framework](https://en.wikipedia.org/wiki/Viola%E2%80%93Jones_object_detection_framework)

[3] [C. Lo, A High-Performance Architecture for Training Viola-Jones Object Detectors](https://tspace.library.utoronto.ca/bitstream/1807/33294/3/Lo_Charles_K_201211_MASc_thesis.pdf)

Hough Transform

[4] [Richard O. Duda, Peter E. Hart, Use of the Hough Transformation To Detect Lines and Curves in Pictures](http://www.ai.sri.com/pubs/files/tn036-duda71.pdf)

[5] [Wikipedia, Hough transform](https://en.wikipedia.org/wiki/Hough_transform)

[6] [Rochester Institute of Technology, Circle Hough Transform](https://www.cis.rit.edu/class/simg782.old/talkHough/HoughLecCircles.html)

OpenCV

[7] [Documentation OpenCV](http://docs.opencv.org/3.2.0/)