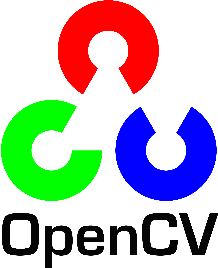
Projet de vision robotique M1

# Choix de l’environnement

Pour notre projet, nous devons faire des traitements sur des images, et pour cela il existe déjà des libraires très puissantes et bien documentée. La plus connue d’entre elles est OpenCV, qui existe depuis de nombreuses années et qui est utilisée dans le milieu professionnel. En effet, la plupart des offres d’emploi incluant du traitement de l’image mentionnent l’utilisation d’OpenCV.

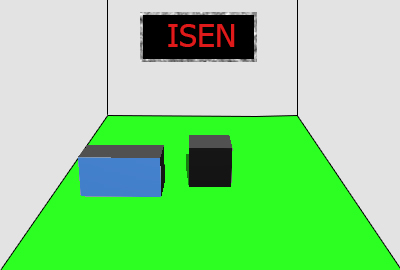


Nous avons donc choisi cette librairie non seulement pour sa puissance et sa robustesse, mais aussi par ce qu’elle était un standard dans l’industrie.

# Détection d’obstacles

Dans le cadre de la V1.0 du projet, nous devons réussir à détecter des obstacles via une caméra dans un cadre « idéal ». C’est-à-dire dans un premier temps avec des conditions d’éclairage avantageuses (éviter les scintillements et les éclairages trop dirigés entrainant des grandes ombres) et dans un environnement au sol de couleur uni.

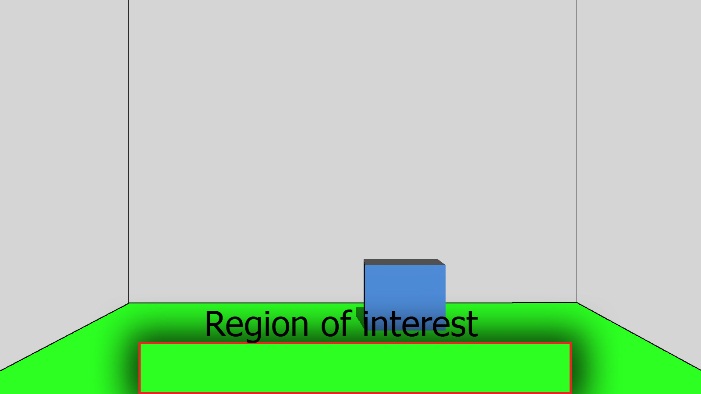
Pour trouver les obstacles, nous choisissons donc de plutôt détecter le sol, et que tout ce qui n’est pas sol est obstacle.



*Exemple d’une salle « idéale » pour le fonctionnement d’une V1*

Pour détecter quelle couleur est le sol, la solution choisie est celle de la mise en place d’une ROI (region of interest), qui permet d’isoler une certaine partie de l’image afin d’y appliquer des traitements ou de s’en servir pour des calculs.

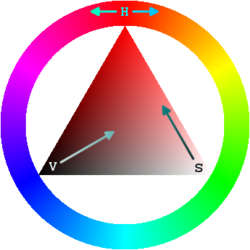
Dans notre cas, la ROI est un rectangle situé en bas de notre image, car c’est dans cette zone que l’on est sûr d’avoir le sol en image, sauf dans le cas où le robot démarrerait collé à un mur.



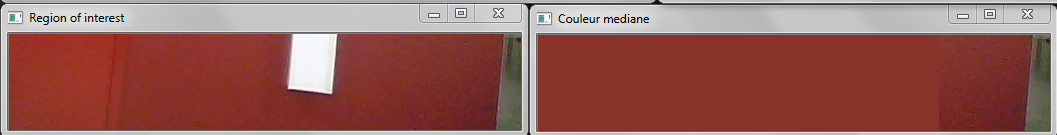
*Exemple de la ROI sur une image*

Pour trouver la couleur à détecter, la première chose essayée était de faire une moyenne sur les trois canaux BGR (format par défaut d’OpenCV), ce qui aboutit rapidement et sans surprise à un échec, car si un défaut (élément de couleur différente) vient se placer dans la zone ROI alors la moyenne ne renvoie plus du tout la bonne valeur.

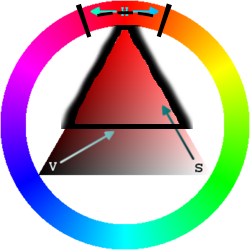
Il a fallu changer deux choses : le système de gestion des couleurs, et la façon de trouver la couleur à traiter.

Pour le premier point, nous sommes passés du BGR au HSV (Blue Green Red, Hue Saturation Value).

Grace au HSV, nous pouvons isoler la teinte (composante H) de la saturation et de la luminosité, ce qui est dans notre cas important étant donné que les conditions d’éclairages peuvent différer. Cela permet donc, en théorie, de détecter une couleur sans tenir compte des conditions d’éclairage.

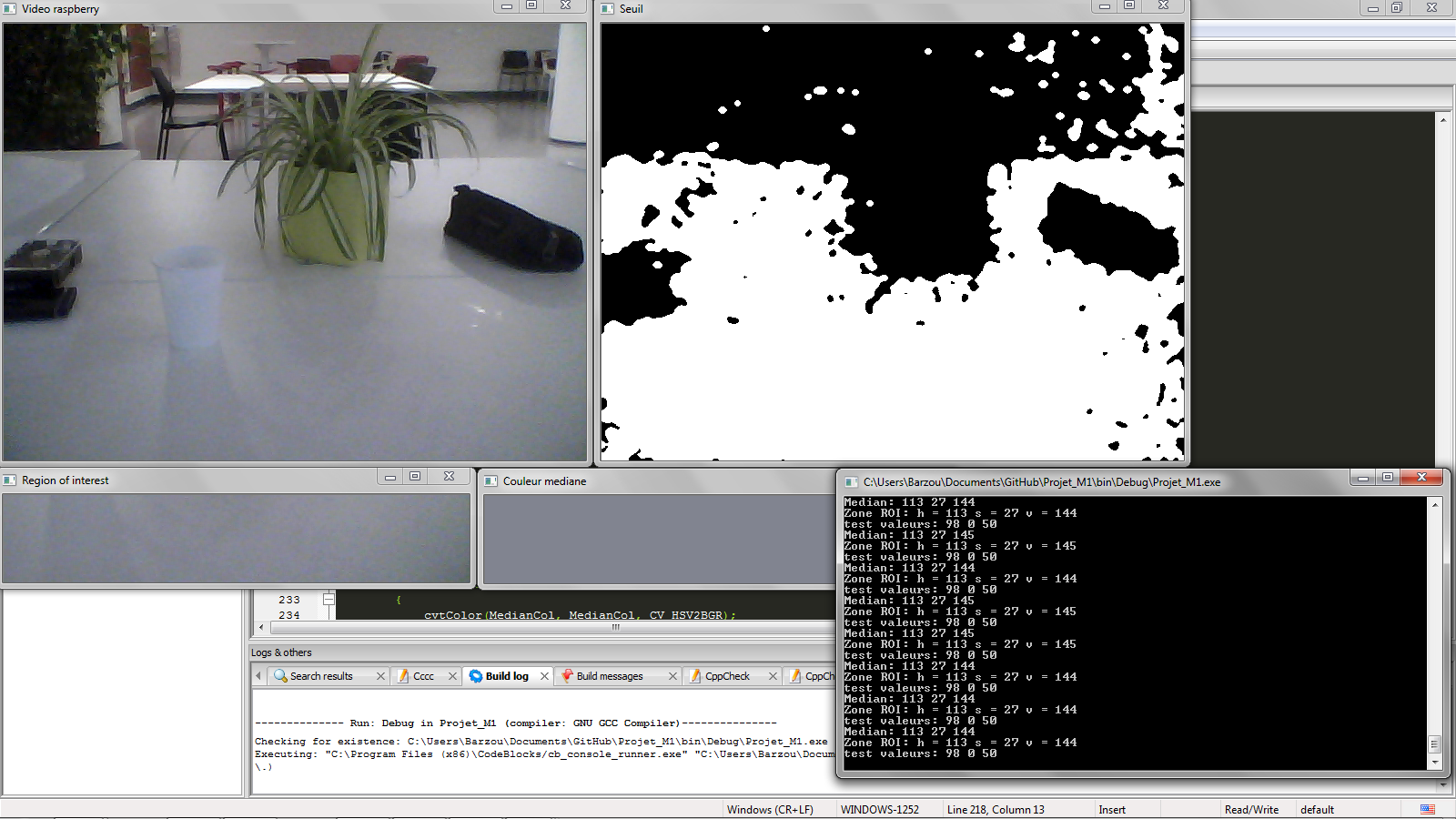
Pour le second point, nous sommes passés à l’utilisation d’une couleur médiane, ce qui permet de trouver la couleur dominante de la zone de pixels sans être affecté par la présence d’une tache par exemple.

*Ici on retrouve bien la couleur rouge du mur malgré la présence de la tache blanche au milieu.*



Une fois la couleur trouvée, nous appliquons un seuillage à l’image entière. Ce dernier se base sur la valeur trouvée via le médian sur la ROI, et même si les valeurs de seuillages sont actuellement en cours de peaufinage, l’idée reste de rester proche de la valeur de la teinte tout en acceptant de prendre des valeurs bien plus variées de la saturation et de la luminosité. Cela permet de garder la sélection de couleur, tout en incluant toutes ses nuances selon la lumière. Un exemple est dessiné sur la roue HSV ci-contre.

Ensuite, via des ouvertures / flou médians afin d’affiner la sélection, nous arrivons à ce résultat :

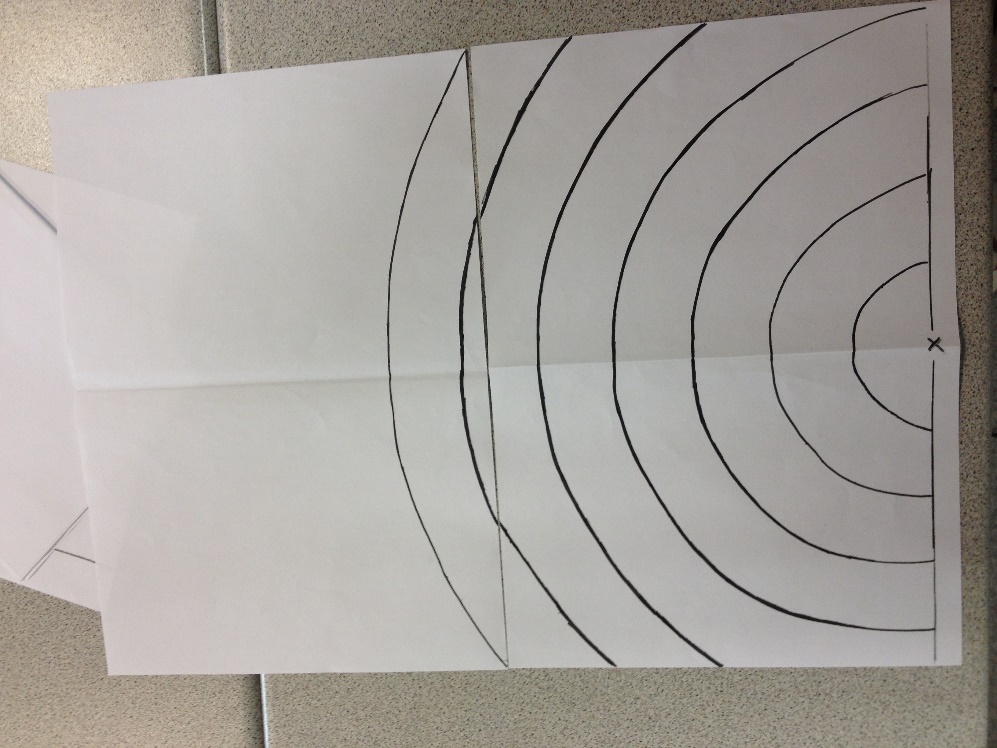


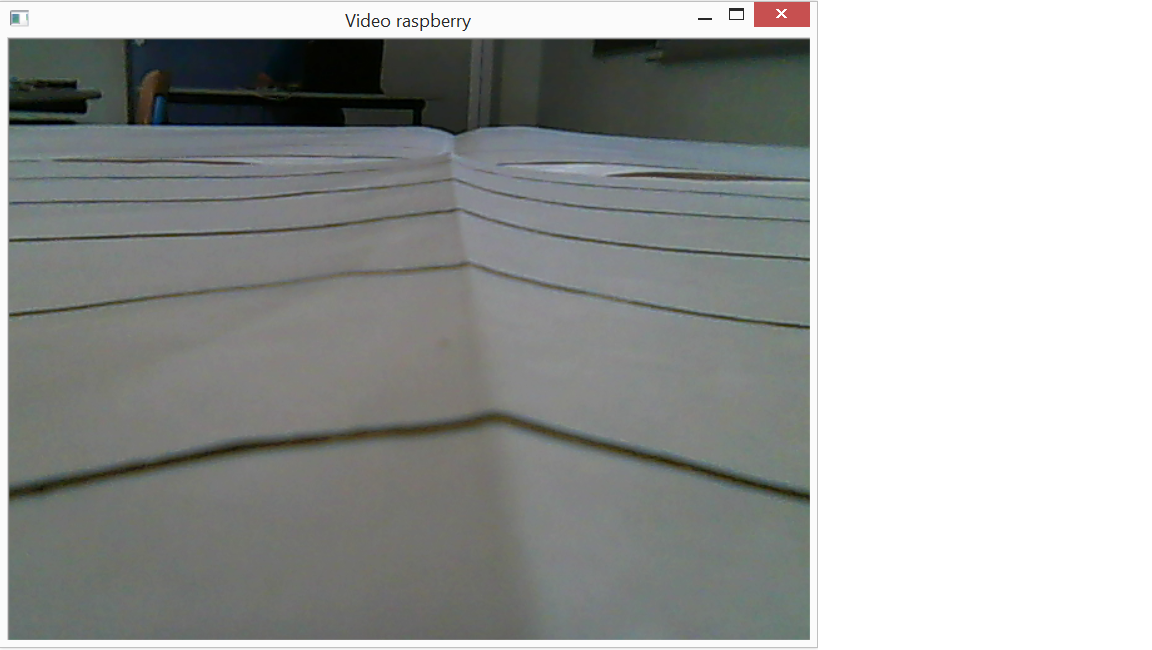
L’utilisation du gobelet blanc est intentionnelle afin de montrer les limites de la sélection par couleur pure, car nous devons être assez tolérants pour éviter d’être sensible aux ombres, ce qui autorise ce genre d’obstacles « fantome ». Toutefois pour une V1, le cadre idéal exclut ce genre d’exceptions.

# Calibrage du flux vidéo

Dans le but d’attribuer une distance à tout obstacle ou chemin que pourrait détecter le robot, il nous fallut définir un calibrage de l’image reçu. Ainsi, après la phase de configuration, tout pixel de la map (format sous lequel opencv récupère l’image) reçue aurait une distance correspondante.

Pour se faire, nous avons établi un mode de paramétrage qui devait faire intervenir le contrôle de l’opérateur, mais qui devait en même temps être le plus simple d’utilisation pour celui-ci.

Nous avons commencé par dessiner un calibre représentant des lignes de distances espacées de 5cm.

La vision d’une caméra placée sur ce calibre ressemble à ça : 

Il est nécessaire d’attendre quelque seconde avant de traiter l’image reçu, car les premières sont trop lumineuse, cela est dû au temps d’adaptation de la caméra à la luminosité présente. Après 2secondes, le programme capture donc l’image et lui applique une transformation en niveau de gris, suivi d’un seuillage afin d’isoler les lignes noires des distances.

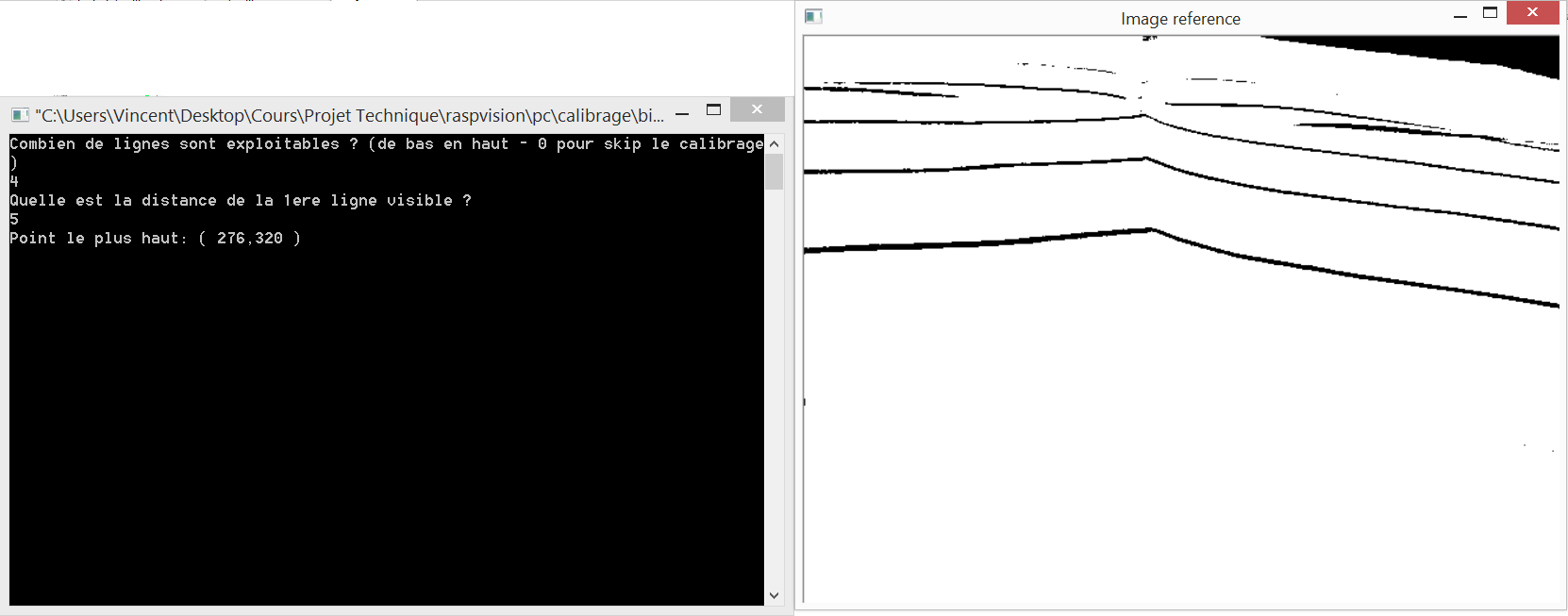
**

Le contrôle opérateur intervient à ce niveau là : il doit vérifier que l’image est exploitable. Le programme lui demande alors d’indiquer combien de ligne sont exploitables (en commençant par celle du bas de l’image), et il doit indiquer la distance à laquelle se situe la première ligne

Le programme de calibrage balaye alors l’image et détecte la ligne noire, il stocke toutes les coordonnées et la distance qui leur est attribuée dans une map<coord\*,int>.

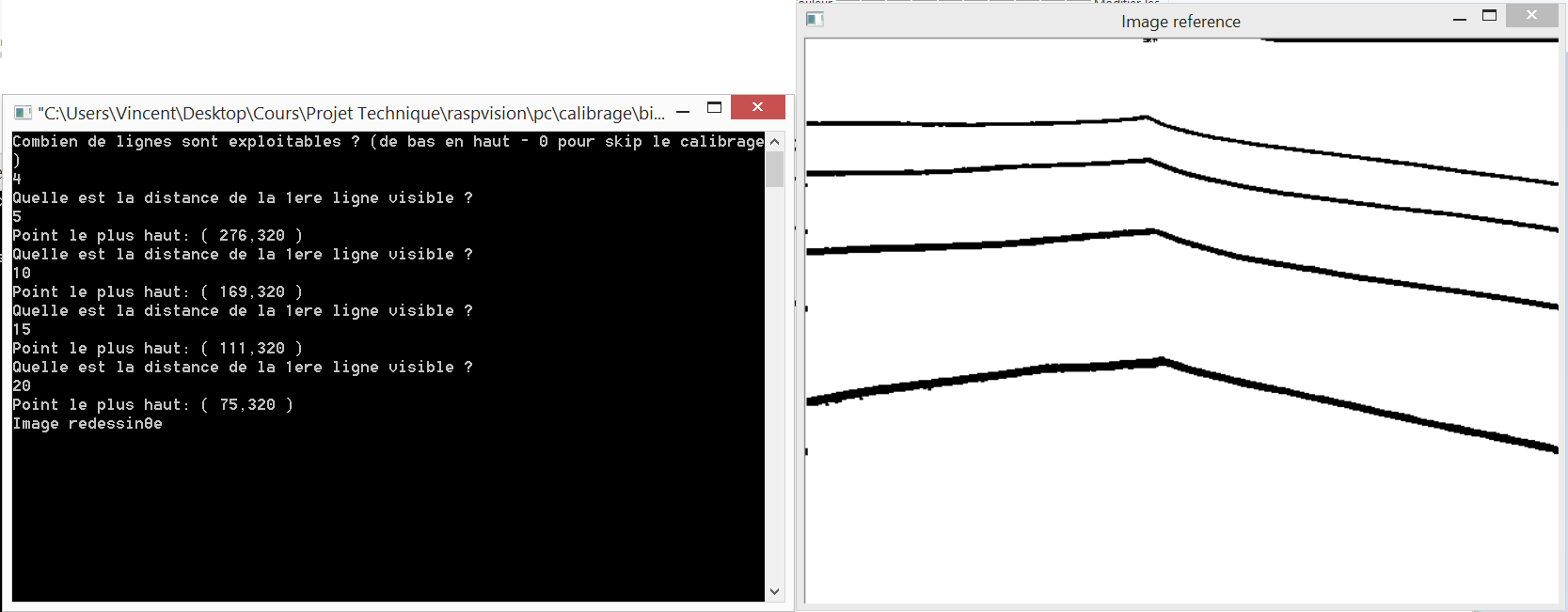
L’algorithme est assez simple : on regarde les pixels d’une colonne en commençant par ceux du bas, si ceux-ci sont blancs, on monte sur la colonne, quand ceux-ci deviennent noirs on commence l’ajout dans la map, enfin si ceux-ci redeviennent blancs c’est que la ligne a été trouvée et on recommence sur la colonne suivante.

Une fois toutes les coordonnées mémorisée, la ligne est effacée de l’image.

**

L’opération se réitère au nombre de lignes exploitables que l’opérateur a indiqué.

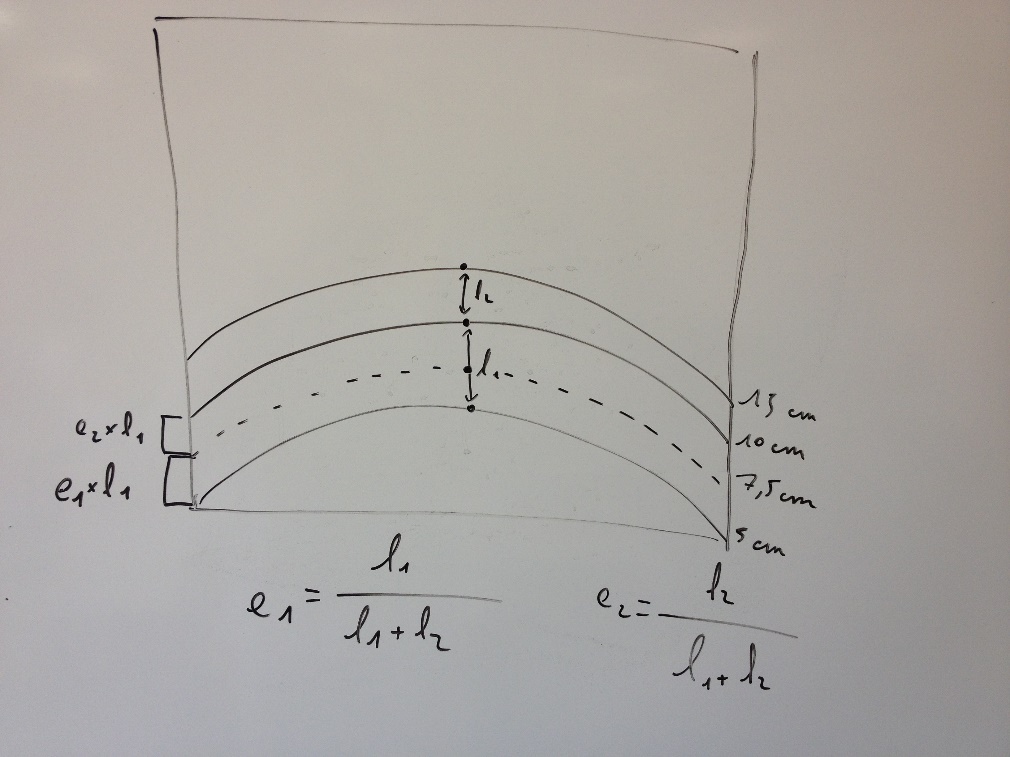
Enfin, une fois le balayage effectué, la frame initiale est entièrement effacée, et on dessine ce qui est stockée dans la map, encore une fois pour validation de l’opérateur. Cela permet aussi d’éliminer certains parasites, notamment sur le haut de l’image.

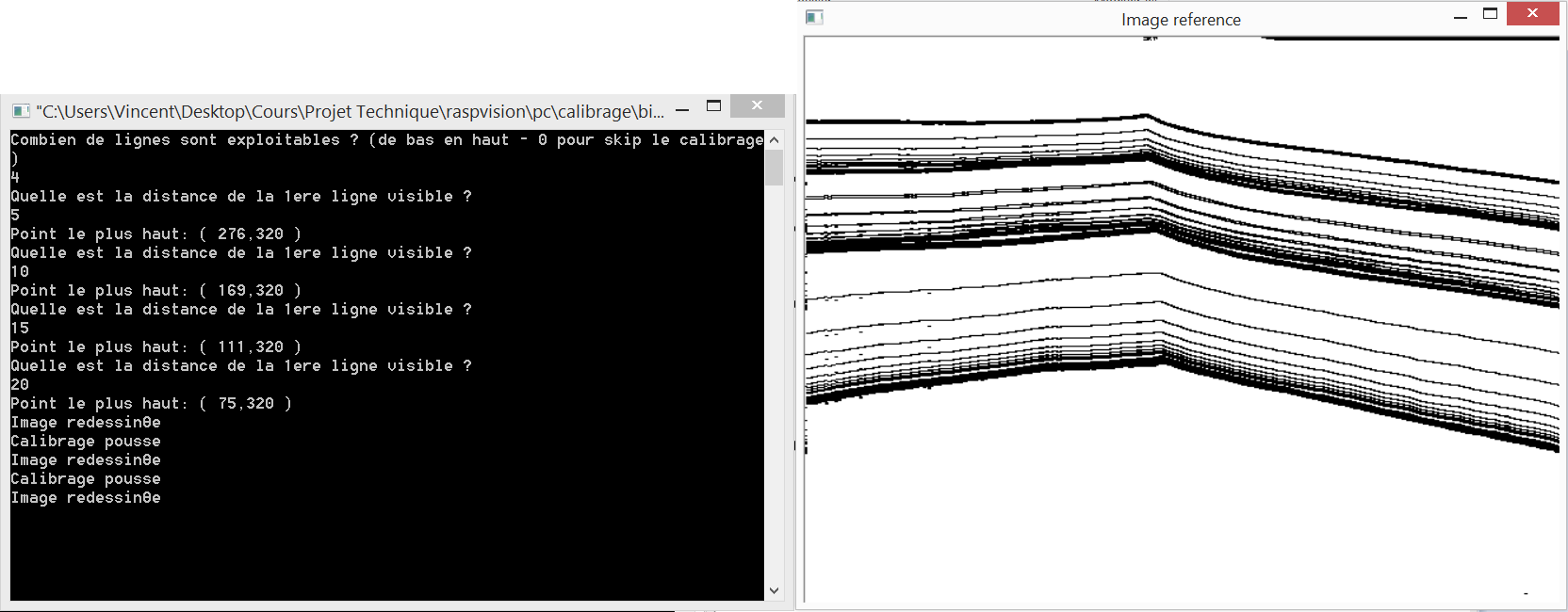
**

Cependant nous n’avons à ce moment-là, que quelques pixels qui ont une distance attribuée. Afin de compléter la map, il nous fallut déterminer plusieurs méthodes possibles.

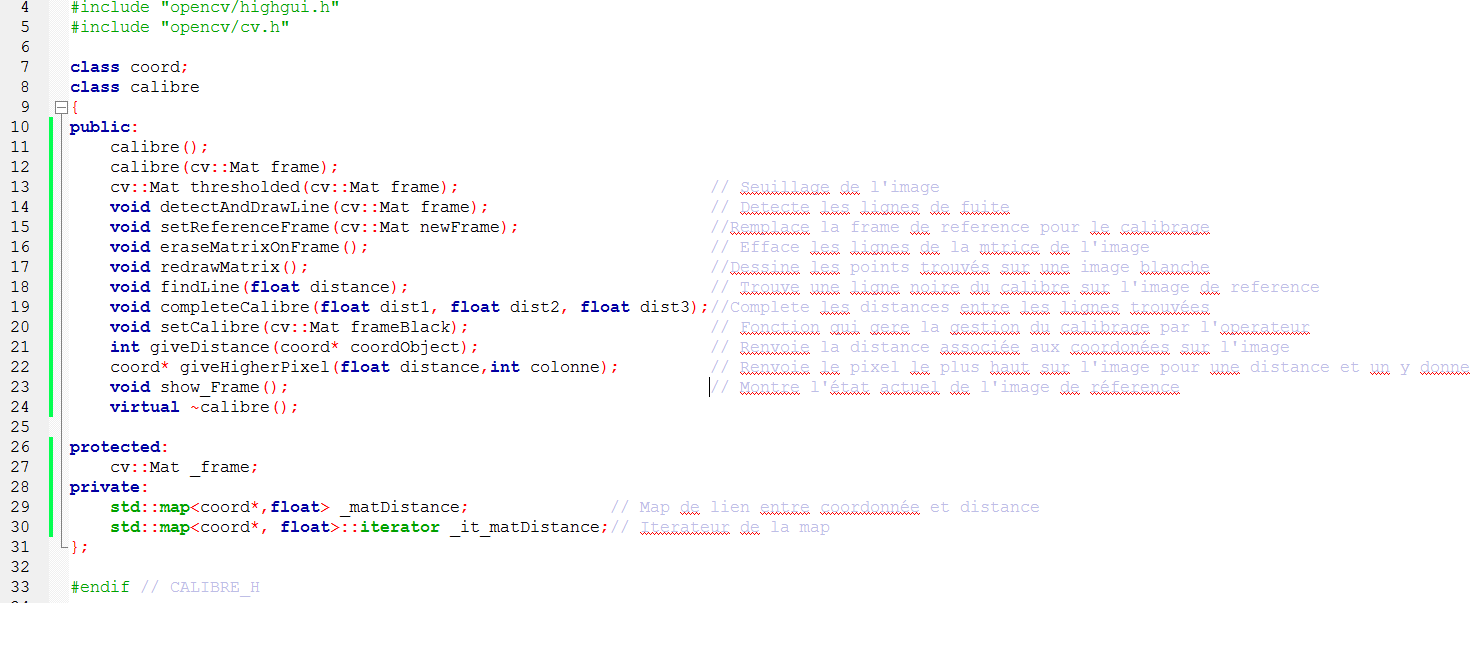
La première à laquelle nous avons pensé fut celle des lignes de fuite : en trouvant le point de fuite de l’image, nous aurions pu tracer ces lignes de fuite et calculer sur ces lignes les distances de chaque point. Mais l’algorithme représentait un travail beaucoup trop lourd, car toutes les recherches que nous avons effectuées nous ramenaient vers des travaux de chercheurs dans le domaine du « vanishing point » (point de fuite). De plus, cette méthode ne palliait pas à tous les cas possible : il aurait été possible d’avoir un point de fuite en dehors de notre image et il nous aurait été impossible de compléter les données.

Nous nous sommes donc penchés vers le facteur d’écrasement. Cette méthode nécessite qu’au moins 3 lignes soient exploitables sur l’image capturée au début. Le principe est de connaitre la différence de pixels entre deux lignes de même distance réelle. Grâce à cela, on peut donc appliquer la proportion trouvée pour tracer de nouvelles lignes et leur attribuer des distances.



Pour l’instant l’algorithme de calcul fonctionne et affiche une ligne intermédiaire (celle en pointillé). Nous rencontrons actuellement des problèmes pour le faire boucler et ainsi combler les blancs de l’image.

Après ça, la partie calibrage sera terminée et opérationnelle.

*calibre.h*