

Linor

Traitemen t d'ima ge



Nathan Latino, Sol Rosca (INF3b)

Mai 2020

Table des matières

1. Introduction	1
2. Objectifs	2
3. Analyse et Conception	3
3.1. Analyse de la vidéo	3
3.2. Techniques utilisées	4
3.2.1. Prétraitement	4
3.2.1.1. Transformation en niveau de gris	4
3.2.1.2. Filtrage du bruit	5
3.2.1.3. Morphologie	7
3.2.2. Segmentation	7
3.2.2.1. Seuillage multidimensionnel	7
3.2.2.2. Détection de contours	8
3.2.2.3. Masque de zone	9
3.2.2.4. Transformée de Hough Probabiliste	10
3.3. Développement	12
3.3.1. Capture de l'écran	12
3.3.2. Traitement de l'image	12
3.3.3. Récupération des données	12
3.3.4. Traitement des données	12
4. Résultats	14
5. Améliorations	16
6. Conclusion	17
7. Références	18
8. Annexes	18

1. Introduction



Une vidéo de présentation se trouve en annexe .

Linor est un projet développé dans le cadre du cours de traitement d'image en 3ème année à l'HE-ARC. Ce projet permet de mettre en pratique la théorie vue dans ce cours.

Le but de ce travail est de fournir une application permettant de piloter un véhicule dans un jeu vidéo. Pour cela, une première partie consiste à traiter une vidéo d'un déplacement en véhicule dans un cas réel. Cette première analyse permet de tester rapidement les types d'algorithmes à utiliser. Le traitement doit fournir une première approche pour trouver la direction de la route.

La deuxième partie du travail est faite sur un jeu vidéo. le flux vidéo du jeu est récupéré et traité en temps réel. Le programme conduit le véhicule grâce aux entrées claviers.

2. Objectifs

- Analyse d'une vidéo enregistrée (cas réel)
 - Réduction du bruit
 - Détection des contours du marquage au sol
 - Interprétation des données
- Application de l'algorithme sur un jeu vidéo
 - Adaptation de l'algorithme au nouveau média
 - Interface permettant au programme de piloter le jeu

3. Analyse et Conception

3.1. Analyse de la vidéo

Dans le cadre de ce projet, de nombreux évènements parasites peuvent complexifier l'acquisition des données. C'est pourquoi il est nécessaire de contourner le gros de ces problèmes pour être sûr de pouvoir les tester lors du développement.

Voici une liste non exhaustive de ce genre évènements:

- Autres véhicules
- Changement de luminosité
- Beaucoup ou peu de marquage
- Effet climatiques
- Reflets
- Lignes non parallèles au marquage
- Dégradation de l'image lors de l'acquisition

3.2. Techniques utilisées

Ce chapitre décrit les différentes techniques de traitement d'image utilisées lors de ce projet.

3.2.1. Prétraitement

3.2.1.1. Transformation en niveau de gris

La transformation en niveau de gris fait perdre de l'information mais apporte plusieurs avantages:

- Réduction du bruit
- Réduction du temps de traitement
- Facilite la mise en place et augmente la précision des systèmes de détection de bord (ex. Canny)

1 | processed_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

Python

`cvtColor` permet de convertir une image d'un espace couleur à un autre.



Figure 1 Transformation en niveau de gris

"luminance is by far more important in distinguishing visual features. John Zhang"

3.2.1.2. Filtrage du bruit

Le filtre de Gauss modifie l'image par une convolution avec un noyau de type gaussien. Il permet d'estomper l'image et ainsi gommer certains défauts.

1 | processed_img = cv2.GaussianBlur(img, kernel, sigmax, sigmay)

Python

`kernel` : La taille du noyau. Donne le nombre de voisin qui seront utilisés dans la convolution. la hauteur et largeur doivent être impaire et positif `sigmax` : Appelé déviation standard. Elle détermine la largeur de la cloche Gaussienne en X `sigmay` : Elle détermine la largeur de la cloche Gaussienne en y. Si elle n'est pas définie, elle prend la même valeur que `sigmaX`

Le noyau choisi est de 5x5. Ce choix est fait après plusieurs tests. Le changement n'est pas drastique entre différents noyaux mais un noyau trop grand ralenti l'exécution (beaucoup de valeurs à traiter). Un noyau trop petit n'atténue pas suffisamment le bruit.

Nous utilisons un noyau dans notre projet. Si le noyau n'est pas valide, la fonction utilisera le sigma. Ce sigma est la taille de la cloche gaussienne en x et en y.

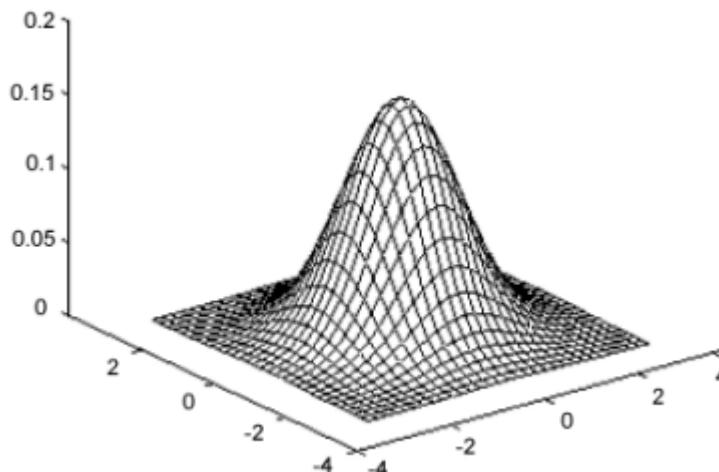


Figure 2 Cloche Gaussienne

Dans notre cas, le filtre enlève entre autre le *film grain* qu'il applique certain jeu pour donner un effet de film ou pour cacher des défauts visuels. Il permet aussi d'atténuer certaines erreurs de rendu du jeu et bruit dans l'image.



Figure 3 grain film on / off

Le filtre gaussien offre un meilleure précision à la détection des lignes après être passées dans un filtre Canny et par la suite avec la transformée de Hough. Il est important de l'appliquer au début du traitement de l'image.



Figure 4 Application du filtre gaussien

3.2.1.3. Morphologie

la morphologie nous permet d'épaissir (thickening) les lignes pour faciliter la détection avec la transformée de Hough. En plus de cela, l'utilisation de l'ouverture morphologique (érosion suivie d'une dilatation) nous permet de supprimer les bruits isolés.

```
1 | size=(3, 3)
2 | iterations=1
3 | kernel = np.ones(size, np.uint8)
4 | processed_img = cv2.erode(img, kernel, iterations=iterations)
5 | processed_img = cv2.dilate(processed_img, kernel, iterations=iterations)
```

Python

`kernel` : donne la taille du noyau à utiliser. par défaut (3x3). `iterations` : le nombre de fois qu'il doit être appliqué.

Nous n'utilisons plus de morphologie dans la deuxième partie du projet car les bruits isolés n'apparaissent plus. Les lignes sont suffisamment épaisses à la sortie de canny pour ne plus avoir besoin de la morphologie.

3.2.2. Segmentation

3.2.2.1. Seuillage multidimensionnel

Le seuillage est une technique de binarisation d'image, il est utile pour trouver certains attributs d'une image. Un seuillage multidimensionnel traite plusieurs images (par exemple dans un format RGB avec les trois canaux de couleur) et retourne une image binaire.

```
1 | low = np.array([h1,s1,v1])
2 | upper = np.array([h2,s2,v2])
3 | hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
4 | processed_img = cv2.inRange(hsv, low, upper)
```

Python

Il est souvent plus précis d'utiliser l'image sous format HSV que RGB. Le format HSV offre la possibilité de seuiller sur la teinte, la saturation et la valeur de la couleur.

Ce seuillage a été utilisé dans la première partie du développement sur la vidéo du cas réel. Elle a été enlevé par la suite. Le jeu vidéo à un système de cycle jour/nuit. De ce fait, la ligne change trop souvent de teinte et de saturation. Le seuillage n'était optimal que à certains moments de la journée.

3.2.2.2. Détection de contours

Pour la détection de contours, le filtre Canny est un bonne outil. L'algorithme permet de réduire encore le bruit de l'image et de retourner une image binaire avec uniquement les contours.

```
1 | processed_img = cv2.Canny(img, threshold_low, threshold_upper)
```

Python

les deux seuils (threshold) s'utilisent de la manière suivante :

- Inférieur au seuil bas, le point est rejeté.
- Supérieur au seuil haut, le point est accepté comme formant un contour.
- Entre le seuil bas et le seuil haut, le point est accepté s'il est connecté à un point déjà accepté.



Figure 5 application du filtre Canny

3.2.2.3. Masque de zone

Pour pouvoir isoler certaines informations, l'utilisation d'un masque est nécessaire. Le masque supprime des régions de l'image.

```
1 vertices = [np.array([ p1, p2, p3, p4], np.int32)]  
2 mask = np.zeros_like(img)  
3 cv2.fillPoly(mask, vertices, 255)  
4 processed_img = cv2.bitwise_and(img, mask)
```

Python

Dans le projet, un polygone permet de limiter la vision pour la recherche des lignes par la suite. Ce polygone en forme de trapèze se situe devant le véhicule.

Le questionnement sur la distance optimale a appliqué nous permet de soulever certains points. Si la hauteur du polygone est trop haute, le taux d'erreurs est plus haut dans la détection de ligne mais permet d'avoir une meilleure prédiction de la direction de la route. Si la hauteur est plus basse, c'est l'inverse. Il faut trouver un juste milieu entre le taux d'erreurs et la prédiction.

Le masque est modifiable pendant l'exécution directement dans l'interface. Cela permet à l'utilisateur de l'adapter au contexte du jeu ou de la vidéo.



Figure 6 Modification de la zone du masque en jeu

3.2.2.4. Transformée de Hough Probabiliste

La transformée de Hough est une technique de reconnaissance de formes. Il a comme avantage d'être rapide (grâce à l'échantillonnage stochastique) en plus d'être résistant au bruit.

```
1 | processed_img = cv2.HoughLinesP(image, rho, theta, threshold, np.array([]), minLineLength, maxLineGap)
```

Python

- `rho` : La distance entre le point en haut à gauche de l'image et le point qui peut être considéré.
- `theta` : L'angle des lignes qui peuvent être considérés (si la valeur est de $\pi/2$ toutes les lignes sont prises)
- `threshold` : Le seuil pour qu'un point puisse être considéré
- `minLineLength` : La longueur minimal d'une ligne pour être validée
- `maxLineGap` : L'espacement autorisé pour considérer une ligne continue

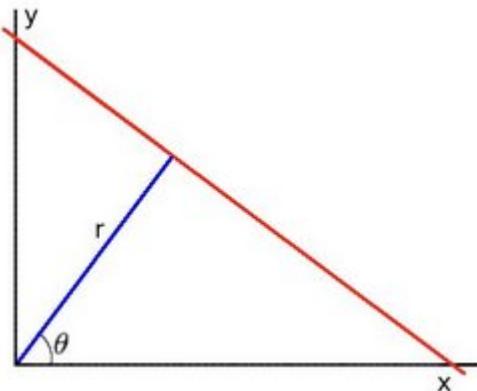


Figure 7 application de Hough

`HoughLinesP` fait une transformé de Hough probabiliste. Cette fonction crée des lignes avec les points données. A la fin, elle somme les points qui forme la ligne. Cela permet d'avoir une probabilité d'un nombre de passage sur une position (pixel). Plus il y a de passages, plus la possibilités d'avoir une ligne à cet endroit est grande.

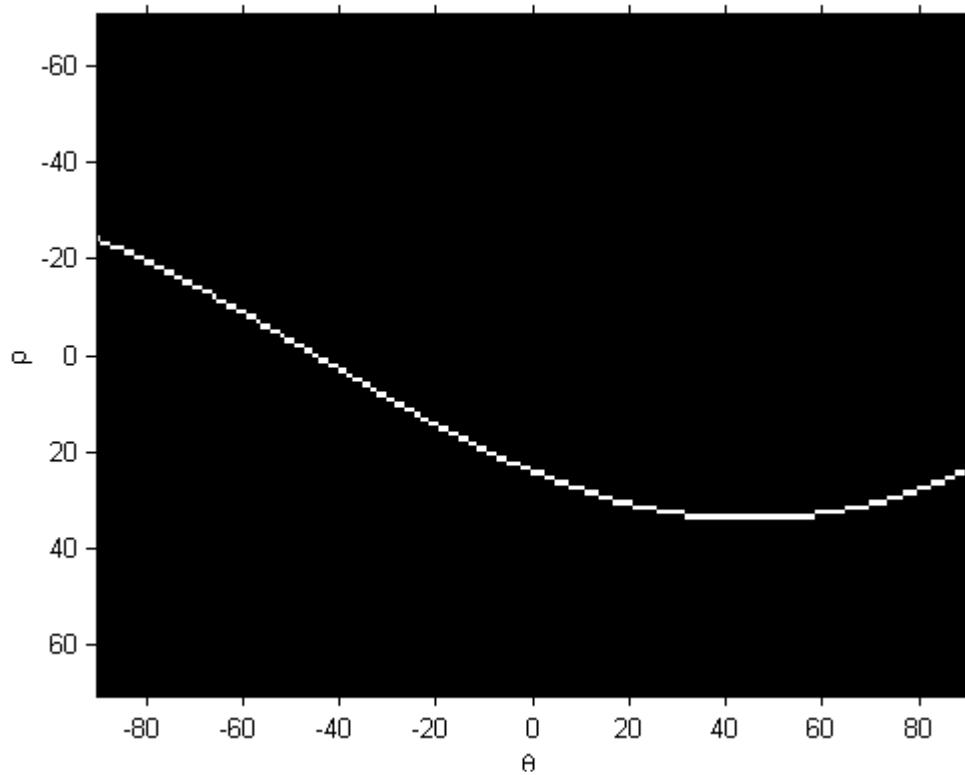


Figure 8 fonction quelconque

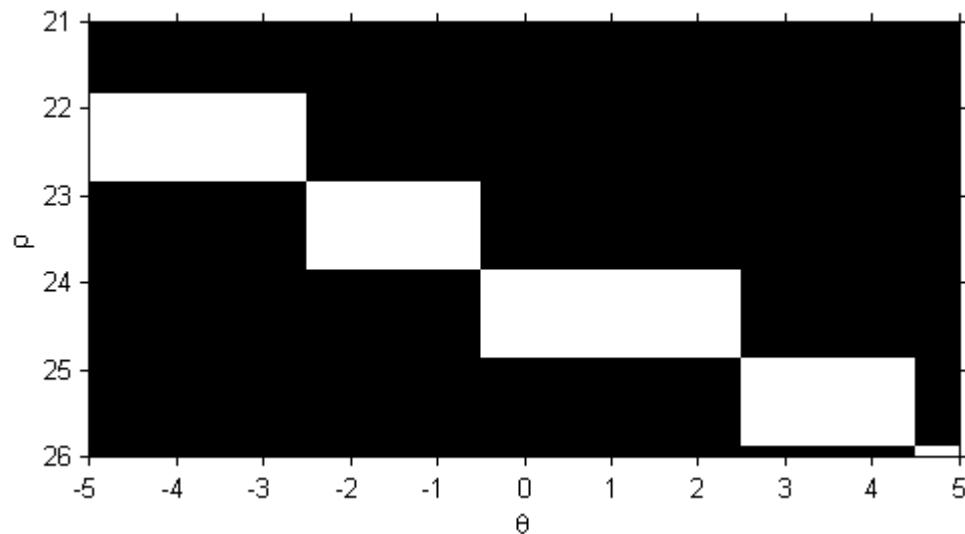


Figure 9 zoom sur la fonction

Pour ce projet, la rapidité d'exécution et la capacité de gérer de nombreuses situations sont des points clés. En effet, le jeu est diffusé en temps réel, il est essentiel que le programme soit réactif. L'algorithme ne doit pas ralentir le traitement de chaque capture du flux vidéo afin d'avoir suffisamment d'images par seconde. Pendant le déplacement en véhicule, de nombreux obstacles ainsi que du bruits peut apparaître. La détection doit être précise et éviter l'ajout de ces lignes parasites.

3.3. Développement

Ce chapitre explique la structure du code et les étapes du traitement des données.

3.3.1. Capture de l'écran

la librairie `win32` est utilisée pour capturer une partie de l'écran. La méthode `grab_screen` permet de prendre une image et de retourner une image en RGB.

3.3.2. Traitement de l'image

L'image passe par différents traitements avant de pouvoir utiliser les informations nécessaire aux calculs.

La méthode `edges` applique les traitements suivants :

- Transformation en niveau de gris
- Filtrage du bruit (`GaussianBlur`)
- Détection de contours (`Filtre Canny`)

Après ces traitements, le **masque de zone** est appliqué avec la méthode `roi`. Les points du polygone se trouve dans la liste de `MASK`.

3.3.3. Récupération des données

Le traitement est terminé, il faut récupérer les données. `find_lines` utilise la **transformé de Hough** pour retourner une liste des lignes trouvées.

3.3.4. Traitement des données

A la fin du traitement, le résultat attendu est un point qui donne la direction à prendre du véhicule. Différents moyens de résolution ont été testés. Voici la liste des étapes :

- Classifier les lignes (gauche, droite, non valide)
- Moyenne des lignes par trame
- Moyenne des lignes sur plusieurs trames
- Calculer le point de fuite

Pour classifier les lignes, il faut identifier leurs positions et leurs pentes. la pente nous permet de connaître le côté du véhicule où elle se trouve. Si la valeur de la pente est positive, la ligne se situe sur la gauche du véhicule et inversement pour une valeur négative. Pour invalider un segment, la distance en abscisse du point A et B ne soit pas égale à 0. Cela enlève les lignes horizontales dont on a plus besoin.

Après la classification, Plusieurs techniques sont possibles pour trouver le point de fuite. Une première technique consiste à faire la moyenne des lignes à gauche et à droite (de manière isolée). Ensuite, elles sont sauvegardées dans un buffer pour avoir une moyenne sur plusieurs trames. Ce buffer permet d'éviter des sauts sur le résultat du point de fuite. Si une erreur intervient dans une trame, elle est réduite par toutes les autres trames précédentes qui avait un résultat juste. Elle permet aussi d'avoir une information continue même si sur une image il n'y a pas de lignes trouvées. Le point de fuite est l'intersection des deux droites moyennées.

Voici les autres techniques:

Moyenne ligne, moyenne des points

- Moyenne des droites sur une trame
- Calculer le point d'intersection
- Sauvegarder le point dans le buffer
- Moyenne des points

Cette technique manque de précision et affiche beaucoup d'erreurs. Le point de fuite n'est pas assez constant. Le point a tendance à avoir une trop grande variance.

Moyenne des points, moyenne des points

- Intersection de toutes les lignes à droites avec toutes celles à gauche
- Moyenne des points sur une trame
- Sauvegarder le point dans le buffer
- Moyenne des points

L'intersection de toutes les lignes ralentissent le processus. Elle crée une chute d'FPS.

La première technique est la meilleure. Les résultats obtenus sont ceux escomptés. Elle ne détériore pas la vitesse de traitement et offre une grande précision. Stocker les droites jusqu'à la fin permet de garder plus de données utilisables. Elle demande plus de mémoire mais ce défaut est très vite compensé par la précision et la rapidité d'exécution.

4. Résultats

Le résultat de la détection est robuste. On peut l'appliquer autant sur une vidéo que dans un jeu vidéo. Malheureusement, le traitement des entrées clavier dépende trop du jeu et de la réactivité des véhicules utilisés.

Le programme a été testé sur plusieurs jeux différents. *Euro Truck Simulator 2* est le jeu principal lors du développement



Figure 10 Flux vidéo



Figure 11 Euro Truck Simulator 2



Figure 12 TrackMania 2 Stadium



Figure 13 TrackMania 2 Valley

5. Améliorations

- Une modification dynamique de la forme et de la position du polygone de détection.
- Une série de tests sur la cohérence des lignes trouvées pour tenter d'éliminer certains faux positifs ainsi que la non discrimination d'une ligne qui malgré qu'elle ne suit pas la tendance est la bonne.
- Ajout de machin learning pour bonifier les résultats des précédents points.
- l'ajout d'un fichier de sauvegarde pour les paramètres de l'utilisateur
- Ajout d'un éditeur pour les entrées claviers (pour pouvoir l'adapter au contexte de l'application qu'observe le programme).

6. Conclusion

la première partie du projet qui a été faite sur une vidéo d'un cas réel a permis de mettre en place les idées pour la suite du projet. Beaucoup de fonctionnalités ont dû être revues pour la seconde partie afin d'avoir une détection plus flexible et fiable.

Les objectifs ont été remplis et le programme permet d'avoir une aide au pilotage de bonne facture. L'ajout d'IA pour la gestion des entrées claviers peut être intéressant pour l'adaptation du pilotage sur différent jeu. Pour le travail sur le traitement d'image, l'algorithme est rapide et permet de jouer avec suffisamment d'image par seconde.

Le résultat dépasse nos espérances et semble de meilleure facture que toutes les inspirations trouvées sur youtube pour nous éduquer au problème. Cela nous motive à le publier pour peut être à notre tour inspirer d'autres développeurs amateurs de conduite automatique.

7. Références

- pythonprogramming.net
- mathworks.com
- [OpenCV-python](#)
- [youtube: TJ Mortimer](#)

8. Annexes

- [linor_guide_utilisateur.pdf](#)
- [linor_presentation.mkv](#)
- [Sources du programme](#)