ensiie
école nationale supérieure d'informatique pour l'industrie et l'entreprise

TP IVA n° 3

Option IRA

Detection, Matching, Registration

Prérequis

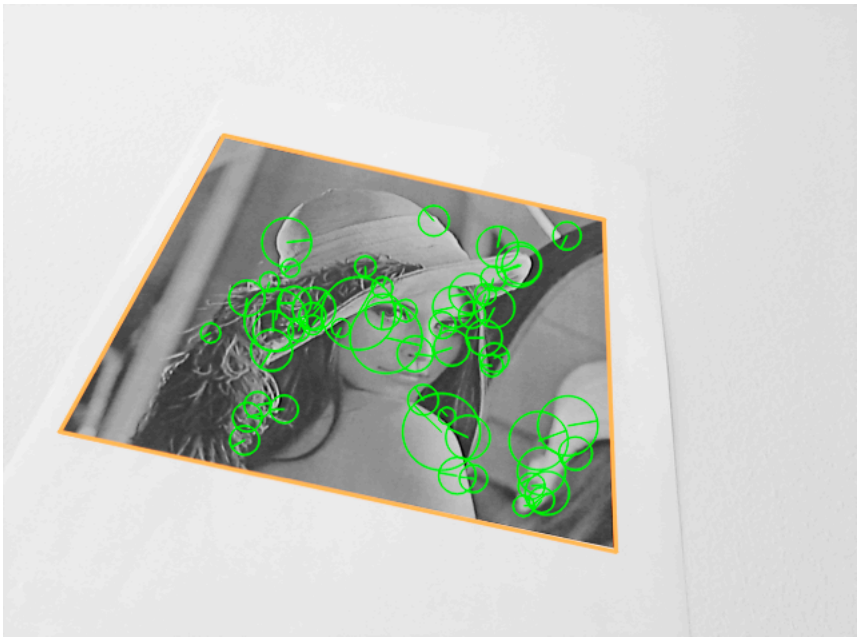
- Chaque exercice se présente sous la forme d'un programme C++ à trous dont vous pourrez trouver les archives dans /pub/IVA/ :
  - Pour les exercices n° 1 & 3: DetMatchReg-<année>-<mois>-<jour>.tgz
  - Pour l'exercice n° 2 : OpenCV_Calibration -<année>-<mois>-<jour>.tar.gz

  Chaque archive contient des sources C++ ainsi qu'un Makefile vous permettant de compiler les programmes. Les programmes fonctionnent en l'état, mais le but de chaque exercice est de compléter ou remplacer certaines parties des programmes.
- Attention toutefois : le programme contenu dans OpenCV_Calibration n'est pas un programme QT, il suffira donc pour le compiler de taper « make » dans le répertoire dans lequel vous aurez décompressé l'archive.
- Documentation OpenCV :
  - En ligne : http://docs.opencv.org/
- Documentation C++ : http://www.cplusplus.com/reference/

1 Feature points

Le but de cet exercice est de calculer des points d'intérêts sur une image modèle (Lena en l'occurrence), puis de rechercher des points d'intérêt dans le flux vidéo et enfin de rechercher un appariement des points d'intérêt du modèle avec ceux du flux vidéo. Lorsque plus de 4 points sont appariés entre le modèle et la scène, on peut alors calculer l'homographie qui transforme les points du modèle vers les points de la scène. En appliquant cette homographie sur les points du modèle et en comparant l'écart avec les points de la scène, on peut alors éliminer les outliers dont la reprojection par l'homographie sont trop éloignés des points de la scène. Les inliers restant pourront alors être utilisés (après calibration de la caméra) pour déterminer la pose de la caméra par rapport au modèle (l'image de Lena imprimée).

Modèle



Scène

1.1 Points d'intérêts

OpenCV permet d'extraire différents types de points d'intérêt en utilisant la série de « factory method » `XXX::create();` des classes filles de la classe `Feature2D` qui permettent d'instancier le « Feature detector » que l'on souhaite sous la forme d'un `Ptr<XXX>`.
Les différents types points d'intérêt que l'on peut utiliser dans ce programme sont :

- FAST : FAST Corner detection d'après [Rosten and Drummond, 2006] : `cv::FastFeatureDetector`.
- STAR : STAR feature, version modifiée des CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching d'après [Agrawal et al., 2008] : `cv::xfeatures2d::StarDetector`.
- SIFT : Scale Invariant Feature Transform d'après [Lowe, 1999] & [Lowe, 2004] : `cv::xfeatures2d::SIFT`.
- SURF : Speeded Up Robust Features d'après [Bay et al., 2006] : `cv::xfeatures2d::SURF`.
- ORB : Oriented FAST and Rotated BRIEF (voir ci-dessous) d'après [Rublee et al., 2011] : `cv::ORB`.
- BRISK : BRISK: Binary Robust Invariant Scalable Keypoints d'après [Leutenegger et al., 2011] : `cv::BRISK`.
- MSER : Maximally. Stable Extremal Regions d'après [Forssén and Lowe, 2007] : `cv::MSER`.
- GFTT : Good Features To Track d'après [Shi and Tomasi, 1994] : `cv::GFTTDetector`.
- KAZE : 風(vent) en hommage à Taizo Iijima, le père des espaces d'échelle (1959), et utilisant un espace d'échelle non linéaire, d'après [Alcantarilla et al., 2012] : `cv::KAZE`.
- AKAZE : Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces d'après [Alcantarilla et al., 2013] : `cv::AKAZE`.

Tous les types de points d'intérêt ont au moins une position, mais les types de points dit « riches » comme les SIFT, SURF ou ORB ou BRISK ont aussi une orientation principale (en général basée sur la direction du gradient, ou la plus forte pente du laplacien), et peuvent aussi avoir une taille (en général déterminée par l'octave et la « note » de l'espace d'échelle dans lequel le point a été détecté initialement). Le tableau ci dessous résume les caractéristiques des différents types de points d'intérêts utilisables.

	Position	Taille	Orientation principale
FAST	✔		
STAR	✔	✔	
SIFT	✔	✔	✔
SURF	✔	✔	✔
ORB	✔	✔	✔
BRISK	✔	✔	✔
MSER	✔	✔	
GFTT	✔		
KAZE	✔	✔	✔
AKAZE	✔	✔	✔

Les points d'intérêt sont modélisés en OpenCV par la classe `KeyPoint` qui contient les champs suivants :

- `Point2f pt` : Position du point d'intérêt.
- `float size` : Taille du point d'intérêt.
- `float angle` : Orientation principale du point d'intérêt
- `float response` : Réponse du détecteur (Mesure de cornerness par exemple pour les points de GFTT ou Harris)
- plus d'autres attributs.

On peut extraire plusieurs types de descripteurs sur les points d'intérêts précédemment obtenus car ils ne sont pas forcément liés[1], néanmoins il n'est pas garanti que si l'on détecte des points ORB par exemple on puisse obtenir un descripteur SIFT qui soit discriminant pour la mise en correspondance entre les descripteurs extraits des points du modèle et les descripteurs extraits des points de la scène.
Les différents descripteurs que l'on peut calculer sur les points d'intérêt sont :

- SIFT : Scale Invariant Feature Transform (vus en cours) adaptés à l'extraction de descripteurs sur les points SIFT. Précis mais lent, basés sur des vecteurs de 128 composantes **valuées**.
- SURF : Speeded Up Robust Features. Précis et plus rapides, basés en g[al] sur des vecteurs de 64 composantes **valuées**.
- BRISK : Binary Robust Invariant Scalable Keypoints descriptor, basé sur 64 composantes **binaires**.
- BRIEF : Binary Robust Independent Elementary Features. Plus rapides car basés sur 32 composantes **binaires** indépendantes.
- ORB : Oriented BRIEF, variation orientée des descripteurs BRIEF développée par l'équipe OpenCV.
- FREAK : Fast Retina Keypoint, basés sur 64 composantes binaires.
- KAZE : basés sur 64 composantes binaires.
- AKAZE ; basés sur 61 composantes binaires.

❖ Etudiez la classe **CvDetector** qui contient le `featureDetector` permettant de détecter des points d'intérêts ainsi que le `descriptorExtractor` permettant de calculer des vecteurs de descripteurs sur chacun des points détectés. Complétez la classe `CvDetector` pour instancier le `featureDetector` (dans la méthode `setFeatureType`) ainsi que le `descriptorExtractor` (dans la méthode `setDescriptorType`) avec les factory méthodes des différent(e)s Feature2D : ***XXX*`::create()`.**
❖ Complétez la méthode update afin de détecter les points d'intérêts en utilisant la méthode **`featureDetector->detect(…);`** et de calculer des descripteurs sur ces points en utilisant la méthode **`descriptorExtractor->compute(…);`**.
❖ Complétez la méthode `drawResults` de la classe **CvDMR** pour afficher ces points d'intérêt dans l'image du modèle et l'image de la scène en utilisant la fonction
  - **`drawKeypoints(…);`**. Le flag **`DrawMatchesFlags::`*`DRAW_RICH_KEYPOINTS`*** vous servira à afficher la taille et l'orientation des points s'ils en possèdent une.
❖ En manipulant le type de feature dans l'interface graphique, déterminez parmi les features :
  - les plus nombreuses ?
  - les plus rapides à détecter un point parmi celles possédant une taille et une orientation ?
  - les plus stables dans un flux vidéo représentant une scène statique parmi celles possédant une taille et une orientation?
❖ En manipulant le type de descripteurs utilisés déterminez :
  - Les descripteurs les plus rapides à extraire sur un point.

1.2 Mise en correspondance des descripteurs

Une fois ces descripteurs calculés il faut les « matcher » en utilisant un «**`DescriptorMatcher`**» que l'on peut instancier grâce à la factory method **`DescriptorMatcher::create(`*`matcherName`*`)`**
dans la classe `CvMatcher` où *`matcherName`* correspond au type de matcher que vous souhaitez instancier.
Le matching pour un descripteur du modèle $X=\left[x_0 \cdots x_{n-1}\right]^t$ à n composantes consiste à rechercher un autre descripteur $Y=\left[y_0 \cdots y_{n-1}\right]^t$ dans la scène dont la distance euclidienne L2=$\left(\sum_{i=0}^{n-1}\left(x_i-y_i\right)^2\right)^{\frac{1}{2}}$, L1=$\sum_{i=0}^{n-1}\left|x_i-y_i\right|$ ou de Hamming[2] est la plus faible.

Un match entre deux descripteurs est symbolisé par la classe **`DMatch`** contenant les champs:

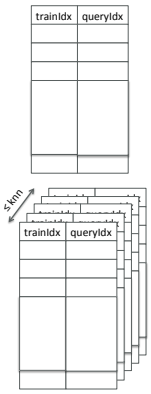
- `int queryIdx` : l'index du descripteur parmi les descripteur de la scène.

[1] Néanmoins l'évolution d'OpenCV tend à coupler fortement détecteurs de points d'intérêts et extracteurs de descripteurs.
[2] Une distance de Hamming est définie comme le nombre de positions où deux suites de symboles diffèrent entre elles (particulièrement bien adaptée aux descripteurs à composantes binaires).

- `int trainIdx` : l'index du descripteur parmi les descripteurs du modèle.
- `float distance` : la distance (L1, L2 ou Hamming) entre les deux descripteurs.

Un descripteur d'indice `trainIdx` du modèle est donc apparié à un descripteur d'indice `queryIdx` de la scène et la distance entre ces deux descripteurs est `distance`.

Le résultat du matching de descripteurs peut être obtenu par différentes méthodes de la classe **`DescriptorMatcher`**:



- **`DescriptorMatcher`**::**`match`** : fournit un **`vector<DMatch>`** contenant l'ensemble des appariements trouvés non-triés et sans bijection entre les deux ensembles (le descripteur n du modèle peut être matché avec les descripteurs k, m & p de la scène par exemple).
- **`DescriptorMatcher`**::**`knnMatch`** : fournit un **`vector<vector<DMatch>`** > contenant les knn (ou < à knn si ce n'est pas possible) meilleurs matchs triés par ordre décroissant de distance.
- **`DescriptorMatcher`**::**`radiusMatch`** : fournit un **`vector<vector<DMatch>`** > contenant l'ensembles des meilleurs appariements triés par ordre de distance qui présentent une distance inférieure à un certain seuil.

Les différents matchers utilisable dans ce programme sont :

- BruteForce : Brute Force Matcher (utilisant une distance euclidienne L2)
- BruteForce-L1 : Brute Force Matcher (utilisant une distance L1)
- BruteForce-Hamming : Brute Force Matcher (utilisant une distance de Hamming)
- BruteForce-HammingLUT: Brute force Matcher (utilisant des distances Hamming dans une LUT)
- FlannBased : Flann based matcher (Fast Library Approximate Nearest Neighbor utilisant une recherche en arbre (k-means trees ou bien k-d trees)) [Muja and Lowe, 2009].
  Le tableau suivant résume les types de matchers utilisables en fonction des descripteurs utilisés : Le programme s'efforcera de ne pas affecter un matcher incompatible avec le descripteur utilisé (quitte à changer celui ci si besoin lors du choix du matcher).

	BruteForce	BruteForce-L1	BruteForce-Hamming	BruteForce-HammingLUT	FlannBased
SIFT	✔	✔			✔
SURF	✔	✔			✔
ORB	✔	✔	✔	✔	
BRISK	✔	✔	✔	✔	
BRIEF	✔	✔	✔	✔	
KAZE	✔	✔			✔
AKAZE	✔	✔	✔	✔	

- ❖ Complétez la fonction **`setMatcherType`** de la classe `CvMatcher` pour instancier le matcher avec la factory method **`DescriptorMatcher`**`::create(`*`MatcherName`*`)`.
- ❖ Complétez la méthode **`update`** de la classe `CvMatcher` pour réaliser un matching simple entre les descripteurs du modèle et les descripteurs de la scène grâce à la méthode `descriptorMatcher->match(…);`.
- ❖ Extrayez les indices de match (**`matches[i]`**`.trainIdx` & **`matches[i]`**`.queryIdx`) qui permettront de créer le sous ensemble de keypoints matchés. Vous pourrez trouver ces instructions dans la section intitulée « Build train and query matched indices vector » après les match dans la méthode update et avant le calcul de l'erreur de match moyenne qui utilise les **`matches[i]`**`.distance` ce qui vous permettra éventuellement de déterminer un seuil applicable à la méthode **`radiusMatch`**. La méthode **`CvDMR`**::**`updateMatcher`** construit ce sous-ensemble de keypoints matchés après la mise à jour du matcher grâce à la méthode **`extractSelectedKeypoints`**.
- ❖ Complétez la méthode **`drawResults`** de la classe `CvDMR` pour afficher ces points d'intérêt matchés dans l'image du modèle et l'image de la scène comme vous l'avez fait précédemment pour les points d'intérêt non matchés.

- ❖ Complétez à nouveau la méthode **update** de la classe CvMatcher pour réaliser un matching croisé (cross matching) entre les descripteurs du modèle et les descripteurs de la scène grâce aux méthodes descriptorMatcher->knnmatch(…) et descriptorMatcher->radiusmatch(…).
  - Le matching croisé consiste à matcher les trainDescriptors aux queryDecriptors (forward matches : matches12) et vice versa (backward matches : matches21) puis à trouver les « match » communs (ceux dont le forward.queryIdx est égal au backward.trainIdx, ou vice versa).
- ❖ Déterminez dans quelles condition il est plus avantageux d'utiliser le Flann Based Matcher par rapport au Brute Force Matcher (En utilisant des points SURF avec des descripteurs SURF par exemple).

1.3 Reprojection des points du modèle dans la scène

Afin de vérifier la validité de la mise en correspondance, nous allons maintenant calculer l'homographie (une homographie est une matrice 3×3 qui modélise la transformation 2D homogène des points d'un plan vers un autre) qui permet de transformer les points de l'image du modèle dans l'image de la scène. On pourra alors calculer les distances individuelles entre les points matchés de la scène et les points matchés du modèle reprojetés dans la scène pour détecter les outliers dans le matching dont la distance est supérieure à un certain seuil (de 1 à 10 pixels) et donc ne garder que les inliers. Cette tâche est réalisée par la classe CvRegistrar.

- ❖ Les KeyPoints matchés obtenus précédemment sont tout d'abord convertis Point2f pour être traités par le registrar car seule la position des points nous intéresse ici.
- ❖ Complétez la méthode **homographyComputing** appelée dans la méthode **update** de la classe CvRegistrar pour calculer l'homographie entre les points du modèle et les points de la scène avec la fonction findHomography(Mat(matchedModelPoints), Mat(matchedFramePoints),CV_RANSAC, reprojThreshold);. Où le seuil reprojThreshold représente la distance maximale entre les points du modèle reprojetés et les points de la scène pour qu'ils soient pris en compte dans le calcul de l'homographie.
- ❖ Une fois cette homographie calculée, on peut alors l'appliquer aux points matchés du modèle pour les (re)projeter dans l'espace de la scène en complétant la méthode **reprojectPoints2D** par l'application de la fonction perspectiveTransform(…).
- ❖ Recherchez dans la méthode **homographyComputing** parmi les points matchés du modèle reprojetés ceux dont la distance est inférieure à reprojThreshold par rapport aux points matchés de la scène pour construire l'index des KeyPoints « inliersIndex » (par opposition aux outliers) qui se reprojettent correctement dans l'image de la scène par homographie.
- ❖ Complétez alors la méthode **updateRegistrar** de la classe CVDMR pour extraire le sous-ensemble des inliers keypoints grâce à la méthode **extractSelectedKeypoints** que nous avons déjà utilisée pour créer le sous-ensemble des keypoints matchés.
- ❖ Complétez ensuite la méthode **drawResults** de la classe CvDMR pour afficher ces points d'intérêt inliers dans l'image du modèle et l'image de la scène comme vous l'avez fait précédemment pour les points d'intérêt non matchés et les points matchés.
- ❖ Pour vérifier la qualité de la reprojection, le plus simple consiste à reprojeter dans l'image de la scène les quatre coins de l'image modèle contenus dans vector<Point2f> modelFramePoints2D puis à dessiner le cadre du modèle reprojeté avec la fonction **plotPolyLine**(…) ce qui est fait à la fin de la méthode **CvDMR::drawResults**.
- ❖ Expérimentations : Déterminez en manipulant les paramètres du **matcher** et de **détection/extraction des descripteurs** (onglet Parameters) les meilleurs paramètres pour assurer un recalage stable et rapide du modèle dans la scène pour :
  - Les SURF (avec descripteurs assortis)
  - Les ORB (avec descripteurs assortis)

2. Calibration de la caméra

Décompressez l'archive OpenCV_Calibration-<année>-<mois>-<jour>.tgz et compilez le programme de calibration d'OpenCV (avec make) pour calibrer la caméra en utilisant la mire (le damier) qui vous est

fourni. Ce damier présente 8 coins internes (points hyperboliques) en largeur et 6 en hauteur (ou l'inverse, au choix). Chaque case du damier fait 30 mm de large.

- ❖ Lancez le programme « calibration » une première fois sans arguments pour obtenir sa documentation (ou voir la fonction `help()`).
  Puis, effectuez la calibration de la caméra en utilisant au moins 10 prises de vues réparties dans l'espace de travail que vous comptez utiliser. Lors d'une « bonne » calibration vous devriez avoir une erreur de reprojection inférieure à 0.5 pixels. Recommencez votre calibration si ce n'est pas le cas.
  La calibration vous permettra de sauvegarder la matrice de la caméra (contenant les paramètres intrinsèques de la caméra (la focale en pixels (f_x, f_y) et le point principal (c_x, c_y)) dans un fichier de type .yml ou .yaml que vous pourrez alors transférer dans le sous-répertoire « cameras » du programme de l'exercice précédent (en modifiant la variable cameraFilename dans le main.c) pour continuer avec l'exercice suivant.

3. Pose de la caméra et recalage

L'utilisation de l'homographie dans l'exercice précédent vous a permis de transformer les points de l'image modèle dans l'espace de l'image de la scène grâce à la fonction `perspectiveTransform`. L'idée est maintenant d'utiliser les inliers de l'exercice précédent combinés avec la matrice caméra obtenue lors de la calibration pour déterminer la pose de la caméra dans l'espace grâce à la fonction `solvePNP`. La fonction `solvePNP` détermine les paramètres extrinsèques (rotation + translation) de la caméra à partir de deux ensemble de points.

- Un ensemble de points 3D dans l'espace de la scène (les points 2D du modèle re-exprimés dans l'espace 3D de l'image du modèle imprimé (avec leur Z à 0.0)).
- Un ensemble de points 2D image leur correspondant dans la scène.

- ❖ Avant de pouvoir rechercher la pose de la caméra, il convient tout d'abord de convertir les points inliers de l'image du modèle en véritables points 3D (correspondant à ces mêmes points sur le modèle imprimé) qui pourront alors être fournis à solvePNP. La taille imprimée de l'image de Lena est de 178.1 mm de large (d'après mon driver d'imprimante), les inliers de la scène doivent donc être mis à l'échelle de la feuille de papier pour former les points 3D de la scène en utilisant la variable `printScale`.
  - Complétez la méthode `CvRegistrar::poseComputing` pour calculer ces points 3D du modèle (on leur donnera une composante z = 0.0) dans inliersModelPoints3D et extrayez aussi les points scene correspondants dans inliersScenePoints2D.
- ❖ Recherchez les paramètres extrinsèques de la caméra avec la fonction `solvePNP` qui vous fournira les vecteurs translation et rotation de la caméra par rapport au modèle imprimé.
- ❖ Reprojetez les points 3D correspondant à la boite englobante de l'image contenus dans la variable `vector<Point3f> modelBoxPoints3D` en complétant la méthode `CvRegistrar::reprojectPoints3D`.
  - Vous pourrez pour ce faire utiliser la fonction `projectPoints` qui reprojette dans l'image des points 3D en fonction des paramètres extrinsèques obtenus avec `solvePNP` et des paramètres intrinsèques obtenus par la calibration de la caméra.

Bibliographie

[Agrawal et al., 2008] Agrawal, M., Konolige, K., and Blas, M. (2008). Censure: Center surround extremas for realtime feature detection and matching. In Forsyth, D., Torr, P., and Zisserman, A., editors, *Computer Vision – ECCV 2008*, volume 5305 of *Lecture Notes in Computer Science*, pages 102–115. Springer Berlin / Heidelberg.

[Alcantarilla et al., 2012] Alcantarilla, P. F., Bartoli, A., and Davison, A. J. (2012). chapter KAZE Features. In *Computer Vision – ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part VI, volume 7577 of Lecture Notes in Computer Science*, pages 214–227. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Alcantarilla et al., 2013] Alcantarilla, P., Nuevo, J., and Bartoli, A. (2013). Fast explicit diffusion for accelerated features in nonlinear scale spaces. In *Proceedings of the British Machine Vision Conference*. BMVA Press.

[Bay et al., 2006] Bay, H., Tuytelaars, T., and Van Gool, L. (2006). Surf : Speeded up robust features. In *proceedings of the 9th European Conference on Computer Vision (ECCV 2006), Part I*, volume 3951 of *Lecture Notes in Computer Science - Image Processing, Computer Vision, Pattern Recognition, and Graphics*, pages 404–417, Graz, Austria. Springer.

[Calonder et al., 2010] Calonder, M., Lepetit, V., Strecha, C., and Fua, P. (2010). Brief: binary robust independent elementary features. In *Proceedings of the 11th European conference on Computer vision: Part IV*, ECCV'10, pages 778–792, Berlin, Heidelberg. Springer-Verlag.

[Forssén and Lowe, 2007] Forssén, P.-E. and Lowe, D. G. (2007). Shape descriptors for maximally stable extremal regions. In *proceedings of International Conference on Computer Vision (ICCV'07)*.

[Harris and Stephens, 1988] Harris, C. and Stephens, M. (1988). A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*.

[Leutenegger et al., 2011] Leutenegger, S., Chli, M., and Siegwart, R. Y. (2011). Brisk: Binary robust invariant scalable keypoints. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2548–2555, Washington, DC, USA. IEEE Computer Society.

[Lowe, 1999] Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision (ICCV'99)*, volume 2, page 1150, Los Alamitos, CA, USA. IEEE Computer Society.

[Muja and Lowe, 2009] Muja, M. and Lowe, D. G. (2009). Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application (VISSAPP'09)*, pages 331–340. INSTICC Press.

[Lowe, 2004] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110.

[Rosten and Drummond, 2006] Rosten, E. and Drummond, T. (2006). Machine learning for high-speed corner detection. In Leonardis, A., Bischof, H., and Pinz, A., editors, *Computer Vision – ECCV 2006*, volume 3951 of *Lecture Notes in Computer Science*, pages 430–443. Springer Berlin / Heidelberg.

[Rublee et al., 2011] Rublee, E., Rabaud, V., Konolige, K., and Bradski, G. (2011). Orb: An efficient alternative to sift or surf. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2564–2571, Washington, DC, USA. IEEE Computer Society.

[Shi and Tomasi, 1994] Shi, J. and Tomasi, C. (1994). Good features to track. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR94)*, pages 593–600, Seattle, WA, USA.

Annexes

Si jamais l'étape de link échoue à cause d'une Feature2D manquante, c'est que le fichier de configuration d'openCV n'a pas été mis à jour pour inclure les xfeatures (SIFT, SURF, etc). Il vous faudra alors créer le vôtre.

1. Créez à la racine de votre compte un répertoire /lib/pkgconfig/ et copiez y le fichier /usr/local/lib/pkgconfig/opencv.pc
2. Editez ce fichier opencv.pc. A l'endroit où vous trouverez `-lopencv_features2d`, ajoutez `-lopencv_xfeatures2d`.
3. Editez votre fichier ~/.bash_profile pour y ajouter la variable d'environnement
   `export PKG_CONFIG_PATH=~/lib/pkgconfig:/usr/local/lib/pkgconfig:$PKG_CONFIG_PATH`
4. Lancez QtCreator et dans votre projet QtCreator (onglets Projets à gauche), vérifiez que la variable d'environnement `PKG_CONFIG_PATH` fait bien partie de votre « Build Environment ». Si ce n'est pas le cas ajoutez la manuellement.