Rapport de PFE : Méthodes de détection par caméra RGB pour un outil interactif en réalité augmentée spatiale

Authors: Gabriel Lorrain, Marc Cerutti, Kevin Durou, Johann Kadionik,
- Responsables: Pierre Benard et Pascal Desbarats
- Encadrants: Philippe Giraudeau et Joan Odicio Vilchez

University of Bordeaux - Computer Science Department - Year 2020-2021 - 4TIN002U - Master 2

Abstract

Le projet hérité de celui de réalité mixte CARDS[GOSR+19], se base sur les méthodes de détection et de tracking par caméra RGB pour un outil interactif en réalité augmentée spatiale. Notre objectif était d'étendre les possibilités sans utiliser de marqueurs. Dans ce but nous avons utilisé une approche proche des techniques de hachage de signal utilisé en électrique, en alternant deux algorithmes de tracking type détection et tracking "pur". Pour une utilisation en temps réel, nos résultats montrent que l'on peut ainsi améliorer la qualité de tracking.

Keywords : réalité augmentée spatiale, réalité mixte, détection, tracking, couleurs.

CONTENTS

1 INTRODUCTION

1.1 Projet CARDS

CARDS[GOSR+19] est un dispositif permettant de combiner des objets physiques et virtuels. Il se veut être un outil numérique, interactif et intuitif, développé par l'équipe Potioc. Utilisant la réalité augmentée spatiale, son but est de permettre des activités collaboratives numériques là où les outils informatiques traditionnels se retrouvent limités. CARDS a par conséquent un intérêt dans le développement de nouveaux aspects éducatifs.

L'installation se compose d'un couple vidéo projecteur/caméra RGB, fixé au dessus d'une table sur laquelle évoluent les objets suivis.

Le principe de base est d'utiliser des marqueurs ArUco[Opea] afin de détecter des cartes physiques en temps réel. Une caméra détecte les cartes, et un projecteur affiche du contenu sur celles-ci, créé depuis Unity3D[Tecb].

L'intérêt de l'extension qui doit être développée est de permettre la détection d'objets et de traquer ceux-ci sans utiliser de marqueurs. La méthode se doit de respecter les contraintes du projet initial et de s'intéresser à au moins un type d'information tel qu'un motif, des couleurs ou d'autres features. La solution proposée doit être également capable d'estimer la position d'un objet dans l'espace 3D.

Au final, ce qui fut réalisé est l'exploration et la mise en oeuvre d'algorithmes basée sur la détection et le tracking de couleur, permettant une preuve de concept pour une implémentation avec des features, implémentée selon une architecture qui nous était libre. Elle devait néanmoins s'inscrire dans l'optique d'une solution permettant de s'éloigner de la méthode de détection et de tracking via les marqueurs AruCo et une implémentation dans Unity3D.

1.2 Contraintes

Ce projet étant l'extension d'un travail précédent, nous avions quelques contraintes fortes pour le mener à bien.

D'abord nous ne pouvions pas modifier le matériel du projet. On ne pouvait ni ajouter de nouveaux éléments physiques (comme une deuxième caméra), ni en déplacer (comme rendre la caméra mobile). De ce fait, nous avons dû nous limiter à du tracking RGB, le seul capteur étant la caméra RGB montée sur le projet.

La seconde contrainte physique est l'accès au matériel pour les tests. Durant cette période de pandémie COVID-19, avoir accès aux locaux de Potioc était impossible. Nous ne pouvions donc pas utiliser de matériel de projection. Nous nous sommes contraints à nous concentrer spécifiquement sur l'implémentation de l'acquisition en utilisant d'autre moyens qu'une caméra, comme des vidéos et des solutions virtuelles depuis Unity.

Nous avions également des contraintes technologiques. De par l'architecture utilisée dans le projet CARDS, nous étions tenus d'utiliser OpenCV comme bibliothèque de traitement du signal vidéo. Nous devions aussi créer un plugin DLL pour Unity se servant de notre travail en OpenCV, grâce aux Native plugins[Teca], dans le but d'être facilement intégrable au projet CARDS. Cependant, dans le cadre des conditions et du temps imparti, nous avions la possibilité de réaliser plusieurs plugins si nécessaires.

1.3 État de l'art

1.3.1 Suivi d'objet

Le suivi d'objet, aussi appelé tracking, consiste à localiser et suivre un ou plusieurs éléments en mouvement dans une vidéo. Cette problématique se retrouve dans une grande variété de domaines, tels que l'interaction homme-machine, la vidéo surveillance, l'imagerie médicale, l'édition de vidéo ou encore, comme dans notre cas, la réalité augmentée. L'objectif principal est donc de localiser l'objet ciblé à travers une suite d'images consécutives. On séparera le tracking dit "pur", qui n'utilise que l'historique de l'objet pour prédire ses mouvements, de la détection, qui fait une recherche au sein de l'image à partir de features pour retrouver l'objet.

Pour améliorer les résultats obtenus, on peut utiliser un outil mathématique très souvent utilisé dans le traitement de signal : le filtre de Kalman[Coh], [Luc]. Il permet d'affiner les résultats obtenus si on a un modèle de mouvement de l'objet, et diminuer les éventuelles erreurs dues au bruit de l'acquisition et de la discrétisation à la fois dans l'espace et le temps. Le principe de ce filtre est en deux phases. D'abord la prédiction de la position de l'objet grâce à son historique, ensuite la mise à jour de la prédiction avec les données réelles mesurées. Ainsi, la précision du suivi augmente avec le temps.

1.3.2 Tracker

OpenCV intègre huit trackers différents qui lui sont propres. Si certains mélangent du tracking et de la détection tel que TLD [KMM12], d'autres se concentrent uniquement sur le tracking, ceux-ci étant alors les trackers dits "purs" [PJS16]. Leur problème réside cependant dans leur difficulté à traquer un objet se déplaçant rapidement ou avec un changement de comportement trop soudain. Ils peuvent essayer dans ce cas de localiser l'objet dans une position plus prévisible et repérer un objet similaire mais différent. On retrouve parmi ces trackers le BOOSTING[YF99], fondé sur l'approche Adaboost (lent et peu fiable), MIL [Bab08] (plus précis mais avec un manque en termes de report d'erreurs), et enfin MedianFlow[ZK10] (plus rapide et efficace avec une meilleure détection d'erreurs).

Il existe aussi les algorithmes Meanshift et Camshift[Opeb], qui utilisent la densité maximale de pixels dans l'image, correspondant à la plus forte probabilité de présence de l'objet. Cela signifie implicitement qu'on utilise une représentation de l'objet dans un autre espace (HSV, histogramme de couleur, MSE, niveau de gris, etc.) qui permet d'extraire uniquement l'objet de l'image. On itère alors pour trouver la probabilité maximale la plus proche en suivant la corrélation du mouvement.

Comme nous avons à disposition une caméra RGB, nous avons pensé qu'implémenter un tracking de couleur simple mais efficace permettrait d'améliorer la précision des trackers OpenCV. Une approche basique du tracking de couleur consiste à isoler et suivre une couleur déterminée. Cette couleur peut-être déterminée par l'utilisateur, par sélection de pixels, ou par détection d'objet.

1.3.3 Détection

La détection permet de cibler un ou plusieurs éléments présents dans une image ou une vidéo. Ces éléments peuvent faire partie d'une classe d'objet, souvent prédéterminée, qui sera alors reconnue. Elle est utilisée dans divers domaines tels que l'annotation d'images, de vidéos, ou encore la détection de visage.

Une méthode de détection est la détection de features. Cela consiste à détecter certaines propriétés pouvant être des structures spécifiques tels que des bords, des coins, des points, ou des couleurs. Il s'agit d'extraire des informations, en "offline" ou en "online", pour chaque image initiale afin de déterminer s'il y a une correspondance dans l'image suivante durant la recherche. Nous retrouvons également parmi les algorithmes de détection de features la détection de coins de Harris [JSS18], un coin étant une région où l'intensité varie fortement dans au moins deux directions. Ce filtre va donc utiliser cette information sur une image en niveau de gris. Le principe est d'utiliser des patchs décalés légèrement les uns des autres dans différentes directions. S'il y a un changement d'intensité dans au moins deux directions, alors il s'agit d'un coin. Il existe aussi l'algorithme de SURF [OR15]. Il est lui-même une amélioration de l'algorithme SIFT [MB17], plus rapide que ce dernier. Il détecte un ensemble de points clés correspondant à des descripteurs, indiquant leur intensité et la direction de ceux-ci. L'idée est d'analyser une image le plus indépendamment possible des caractéristiques, telles que l'échelle ou la luminosité. Enfin, nous pouvons citer ORB [ERB11], un algorithme open-source, contrairement à SURF, calculant moins de points clés que ce dernier, mais qui sont tout aussi efficaces. Cet algorithme utilise les techniques de FAST et de BRIEF. Il trouve des points clés grâce à FAST[RHGS15], et applique la détection de coins de HARRIS. Il complète son approche par une redirection de l'algorithme de BRIEF [MCF10] selon l'orientation des points clés et calcule les descripteurs de l'image.

1.3.4 Tracking vs Détection

Les deux approches ont des principes de base bien différents, et ainsi les résultats de ces solutions peuvent être plus appropriés dans certaines situations, même si les deux ont le même objectif : le suivi d'objets.

Le tracking d'objet permet de mieux suivre des types d'objets bien différents sans a priori sur son type, en utilisant la propriété que les mouvements sont continus et assez prévisibles. Cette solution permet aussi de facilement garder un suivi et d'identifier les objets, du moment qu'ils sont correctement initialisés. Le tracking par contre utilise de la mémoire pour pouvoir prédire les futurs mouvements, et ces solutions échouent dans des cas particuliers comme les mouvements brusques ou les occultations, sans pour autant se rendre compte de la perte de l'objet, ce qui est problématique.

La détection permet de combler les défauts cités précédemment. Comme elle recherche des correspondances de l'objet dans l'image, on peut ainsi avoir un seuil de tolérance et filtrer les solutions proposées. Ces solutions se basent cependant sur des informations propres à l'objet, qui sont connues (ex: reconnaissance de visage) ou apprises (ex: extraction de features). Cela suppose que ces solutions sont perdues si deux objets similaires existent dans la même image. Ces solutions de détection sont souvent plus coûteuses que le tracking pur dans des cas plus complexes que la détection de couleurs, et où l'on veut l'estimation de la position et le suivi d'objets.

Des approches existent déjà, essayant de mixer les deux comme le tracker TLD de OpenCV. Mais ce dernier fait beaucoup trop d'erreurs de faux positifs, qui le rend inutilisable dans notre cas[Mal], [Hon].

1.3.5 Approche deep learning

Au cours des années 2010, on assiste à un véritable engouement pour le développement d'algorithmes de détection et de tracking d'objets basés sur le deep learning (ou apprentissage profond en français). L'objectif est alors d'améliorer significativement les performances et la précision de ces méthodes.

Un des algorithmes précurseurs en matière de détection statique est R-CNN[GDDM13] (2013). Le principe est de d'abord découper l'image en plusieurs régions d'intérêt contenant potentiellement des objets, en général par recherche sélective. Ces régions sont converties en vecteurs de taille fixe et sont passées en entrée d'un réseau de neurones convolutif dont le modèle est décrit par Krizhevsky et al.[KSH12] (2012). Ce réseau de neurones permet alors d'extraire les caractéristiques des régions, qui sont finalement classifiées en utilisant un SVM pour chaque classe. Ces trois étapes sont indépendantes mais coûteuses en temps d'exécution car leurs composantes doivent être entraînées séparément. L'architecture de R-CNN a ainsi été retravaillée par plusieurs autres équipes de recherche dans le but de le rendre applicable à la détection en temps réel. On peut notamment citer Faster R-CNN[RHGS15] (2015). Des approches différentes de détection en temps réel ont depuis été proposées, comme You Only Look Once[RDGF15] (2015) et sa version plus récente YOLOv4[BWL20] (2020). Un unique réseau de neurones prédit les boites englobantes des objets de l'image puis les classifie : puisque le pipeline de détection y est unifié contrairement à R-CNN, les performances s'en retrouvent grandement améliorées.

Parallèlement aux méthodes de détection, le deep learning a également été appliqué aux algorithmes de tracking d'objets depuis quelques années. GOTURN[HTS16] (2016) est ainsi un des premiers algorithmes de tracking générique d'objet individuel incorporant un réseau de neurones et capable de suivre un objet à 100 images par seconde. Il fait partie des algorithmes basés sur les *Deep Regression Networks*. Chaque image d'un flux vidéo, ainsi que celle qui la précède, est coupée et recentrée sur la dernière position connue de l'objet suivi. Celles-ci sont ensuite passées en entrée d'une succession de couches de convolution indépendemment l'une de l'autre, dans le but d'en déterminer les caractéristiques à un haut niveau. Les vecteurs obtenus en sortie sont concaténés et passés en entrée de couches entièrement connectées afin de les comparer et finalement déterminer les coordonnées de la boite englobante de l'objet suivi sur l'image actuelle. Le réseau de neurones est lui-même entraîné hors-ligne sur un grand jeu de données annoté avec les boites englobantes des objets (mais pas leur classe puisque l'algorithme doit rester générique).

Bien que GOTURN soit efficace, cette méthode reste limitée à du tracking individuel d'objet. Pour suivre plusieurs objets en même temps, il faut utiliser d'autres approches qui sont souvent basées sur d'autres algorithmes. Par exemple, Recurrent YOLO ou ROLO[NZH+16] (2016) est une extension de l'algorithme de détection d'objets YOLO[RDGF15] mentionné précédemment. Les bounding boxes de chaque objet, obtenues grâce à YOLO, sont concaténées à un vecteur de features obtenu en passant l'image en entrée d'un réseau de neurones convolutif traditionnel. Le tout est passé dans un réseau de neurones récurrents, plus précisément une cellule *Long short-term memory*. L'objet peut alors être suivi par régression de ces informations spatiales et temporelles.

Certains algorithmes de tracking utilisant le deep learning sont également basés sur des algorithmes de tracking classiques. DeepSORT[WBP17] (2017) est ainsi une extension de l'algorithme de tracking sans deep learning SORT[BGO+16] (2016). Ce dernier combine des méthodes connues comme le filtre de Kalman et l'algorithme hongrois[KY55] pour effectuer le suivi d'objet. S'il est rapide et assez précis, SORT a cependant le défaut d'être peu robuste aux occultations. C'est dans le but d'améliorer cette robustesse que DeepSORT intègre un simple réseau de neurones convolutif dont le but est d'évaluer une nouvelle métrique basée sur l'apparence de l'objet. Les tests effectués par les auteurs de DeepSORT affichent une réduction de près de moitié des changements d'identité des objets suivis par rapport à SORT. La simplicité d'implémentation de DeepSORT et sa capacité d'être utilisé en temps réel en a fait un des algorithmes de tracking avec deep learning les plus populaires et répandus.

Pour conclure ce tour d'horizon des algorithmes de tracking utilisant des réseaux de neurones, on peut également citer Real-time MDNet[JSBH18] (2018), une amélioration de l'algorithme de tracking MDNet[NH15] (2015), qui est inspiré de l'algorithme de détection Fast R-CNN[Gir15] (2015), tout comme MDNet était inspiré de R-CNN[GDDM13]. Real-time MDNet est capable comme son nom l'indique de suivre les objets en temps réel, alors que MDNet est limité par son importante complexité spatiale et temporelle due au fait que chaque région de l'image proposée est traitée individuellement.

2 SPÉCIFICATIONS

2.1 Analyse des besoins

2.1.1 Fonctionnels

- **Détecter dynamiquement des objets sans marqueurs.** L'objectif principal est de pouvoir détecter des objets sans marqueurs type ArUco et sans qu'ils n'aient été nécessairement enregistrés manuellement par l'utilisateur. Cela implique d'explorer différents types d'algorithmes de détection.
- **Suivre un objet sans marqueurs.** Le projet doit être capable de suivre un objet sans marqueurs pendant l'exécution du programme.
- **Estimer la position en 3D d'un objet.** Nous devons pouvoir estimer la position de l'objet dans la zone de travail, ce qui passe entre autre par la calibration de la caméra.
- **Intégrer dans Unity3D.** Nous devons permettre l'intégration à Unity, c'est-à-dire faire une bibliothèque DLL avec le cœur des fonctionnalités et les composants dans Unity permettant leur exploitation.

2.1.2 Non-fonctionnels

- **Performance.** Le workflow du projet doit être suffisamment efficace pour être capable de traiter les données en temps réel. Cela revient à tourner à environ 30 FPS avec tous les calculs de Unity derrière (scripts, physique, animations, etc).
- **Robustesse.** La détection et le tracking d'objets doivent être robustes aux changements de conditions d'éclairage et aux occultations. De plus il doit pouvoir suivre des objets qui ne sont pas forcément uniformes entre eux et éviter au mieux les pertes de tracking, et au pire pouvoir retrouver les objets.
- **Développement, tests et mesures.** Le plugin DLL pour Unity doit pouvoir être chargé et mis à jour à la volée pendant l'exécution. On doit aussi pouvoir extraire les données facilement de mesure de qualité, et de performances. Pour des raisons de comparaison, cela revient à faire un système d'annotation de vidéos, de comparaison d'objets virtuels, et de mesure d'erreur.
- **Maintenabilité.** L'architecture du projet doit permettre d'implémenter facilement des algorithmes de détection et de tracking d'objets différents de ceux actuellement employés, et le projet doit être compatible et intégrable au système CARDS.
- **Licence.** Le projet doit être sous licence *Open Source* pour permettre la poursuite du développement par nos encadrants.

2.2 Scénarios d'utilisations

Le principal cas d'utilisation prévu par notre projet est de tracker des objets de couleurs uniformes, uniques, relativement plats, et qui se démarquent du background afin de permettre leur détection. On s'est concentrés sur des objets plutôt plats car étant plus faciles à tracker que des objets 3D qui seraient sujet à des déformations du fait de la perspective, mais aussi aux ombres et aux rotations lors de leur manipulation.

Ce cas idéal est cependant loin des ambitions du projet CARDS, mais il permet déjà une grande variété de scénarios, comme du brainstorming mixte virtuel/papier, de la reconnaissance de jeux de cartes, ou encore des ateliers éducatifs d'apprentissage de classification d'images pour les enfants par exemple. Nous avons aussi prévu des cas où les conditions de départ changeraient, comme la luminosité principalement.

Une grande thématique qui est cependant traitée superficiellement car complexe serait les occultations lors de la manipulation des objets, car le principal cas d'utilisation impliquerait d'être fait par plusieurs personnes en simultané, et donc que les objets se croisent.

2.3 Architecture

Le projet est réparti en plusieurs parties comme montré figure 1. On peut y voir notamment la séparation entre les différents exécutables, et modules en leur sein. Le centre des fonctionnalités est partagé entre Unity et le DLL,

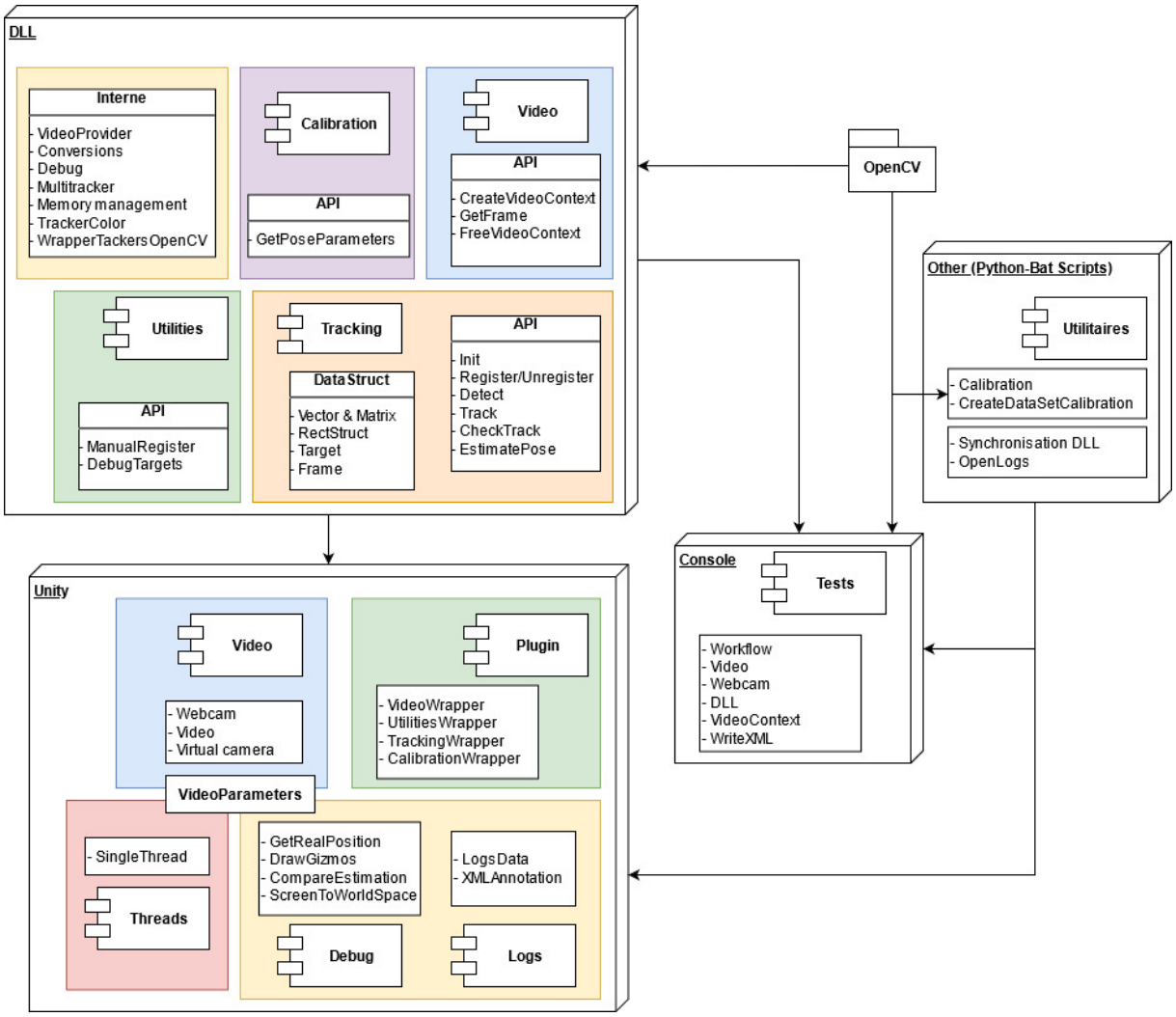

Fig. 1. Architecture du projet et modules.

3 Implémentation

3.1 Environnement

L'environnement de travail que nous avons utilisé a été en grande partie dicté par les contraintes liées au projet CARDS (1.2). Nous devions donc faire en sorte que nos algorithmes de tracking puissent être utilisés par Unity, et pour les besoins de test et démonstrations, nous avons dû recréer un projet Unity reprenant les principes du projet CARDS. Une partie du projet est donc codé en C# pour la logique d'utilisation de nos algorithmes, de l'affichage des résultats et de l'analyse des performances.

La bibliothèque renfermant l'API du cœur de notre solution est quant à elle développée en C++. Ce choix a été en grande partie dicté lui aussi par les impératifs du projet CARDS. Nous devions impérativement utiliser OpenCV comme bibliothèque de référence pour le traitement d'image. Elle dispose de wrappers vers de nombreux langages, comme Python, Java ou Javascript, cependant, pour créer des DLLs utilisables par Unity, le langage C++ est le plus pratique de ceux proposés par OpenCV. Par ailleurs, c'est un langage qui nous est familier et c'est le langage natif de cette bibliothèque. C'est donc dans ce langage que l'on a développé le coeur du projet, et fait la documentation en conséquence.

Pour faciliter le développement et le debug, nous avons mis en place deux scripts .bat (environnement Windows) utilisant un script python personnalisé pour pouvoir facilement mettre à jour les fichiers DLL dans le projet Unity. Chacun des ces scripts est chargé de mettre à jour les liens symboliques du projet Unity pour chaque version du projet (Debug ou Release). Il s'agit en fait de parcourir une liste des fichiers DLL préalablement établie en fichier texte et d'actualiser les liens symboliques dans la version du projet Unity concernée. Ce workflow nous a permis d'être plus efficaces et d'avoir une installation avec moins d'erreurs.

Ensuite, il s'est avéré que nous avions besoin d'un petit utilitaire pour annoter rapidement des vidéos afin de mesurer l'efficacité des trackers, ainsi que d'un programme plus léger que Unity permettant le debug

du DLL. Nous avons choisi d'implémenter un programme console en C++ regroupant quelques tests, et de continuer à utiliser OpenCV pour l'implémentation des annotations. La bibliothèque dispose de tous les outils nécessaires et est déjà installée pour la DLL et donc nous pouvions l'utiliser pour les fonctions de lecture/écriture de fichier XML afin de comparer les annotations aux résultats des algorithmes.

Enfin, pour l'estimation de position nous avons eu besoin de la calibration de la caméra. Là encore, des utilitaires en python ont permis de répondre à ce besoin, permettant une plus grande liberté sur d'éventuelles modifications permettant d'inclure d'autres informations.

3.2 Algorithmes

3.2.1 Général

Comme vous pouvez le voir figure 2, l'algorithme général que reflète notre API est en 3 étapes, sans l'initialisation et la libération de la mémoire.

On fait tout d'abord une phase de détection périodique afin d'enregistrer les nouveaux objets à tracker. C'est aussi durant cette étape que l'on peut demander à l'utilisateur des interactions sur la gestion des trackers.

Ensuite la seconde phase est celle du tracking en soi, où on fera périodiquement de la détection d'objets d'un point de vue algorithmique, **CheckTrack**, et sinon on fera un algorithme de tracking pur, **Track**.

Enfin on fait l'estimation de pose à chaque frame.

Ces étapes peuvent être en partie paralléliser.

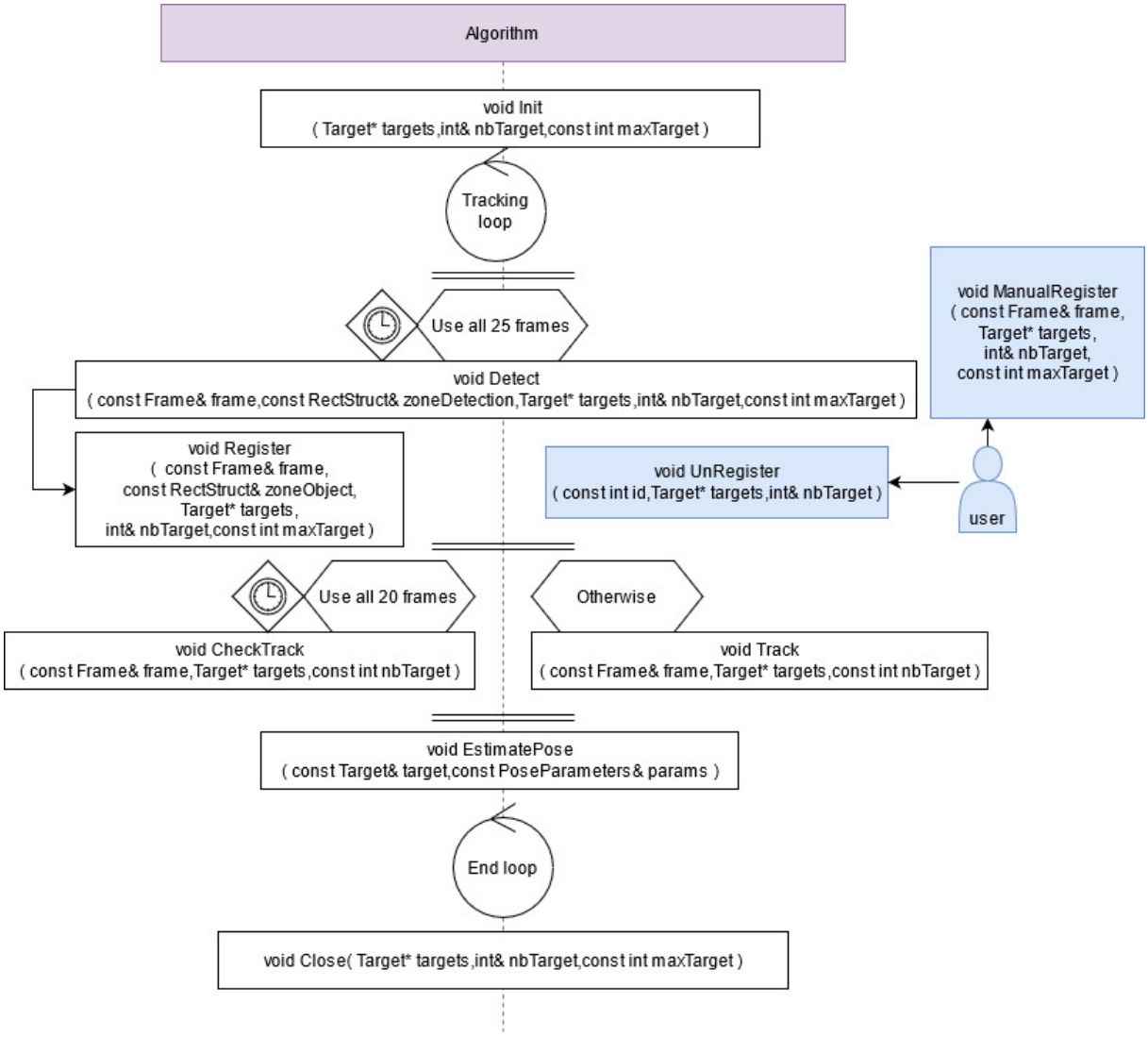

Fig. 2. Workflow général de l'algorithme.

Pour ce projet, nous avons fait le choix de faire l'impasse sur l'implémentation d'un algorithme basé sur le deep learning car cela nécessite une certaine installation et implémentation particulière depuis Unity (chargements des réseaux et modèles pré-entraînés) et nous n'avions pas le temps d'implémenter ces méthodes. Nous n'avions que deux mois pour réaliser un prototype et aussi il aurait été compliqué d'entraîner le réseau de neurones tout en sachant que nous devions implémenter en parallèle des algorithmes de tracking basés sur la couleur et sur les features. C'est pourquoi nous nous sommes limités à des méthodes plus basiques afin d'obtenir un prototype fonctionnel en fin de projet. Pour ces mêmes raisons de temps, nous n'avons pas implémenté l'algorithme de Kalmann.

3.2.2 Détection d'objet

Pour la détection d'objet nous avons décidé d'utiliser le MSE pour détecter un objet dans une zone prévue à cet effet. L'algorithme utilisé passe en niveau de gris la zone de détection sur la frame utilisée comme référence d'arrière-plan et la frame actuelle, puis la différence binaire entre les deux images. On nettoie le bruit grâce à l'érosion ce qui nous permet d'obtenir exclusivement l'objet détecté en pixels blancs, puis on effectue les opérations suivantes nécessaires au MSE à partir de cette différence. Le MSE est normalisé. Si le MSE est supérieur à une valeur limite, alors nous considérons que l'objet est détecté. Nous utilisons OpenCV pour trouver les contours de l'objet et entourons ceux-ci avec une boîte englobante, laquelle est utilisée pour enregistrer les informations nécessaires pour le tracking.

Nous vérifions ensuite s'il y a une place en mémoire disponible, car du fait du partage mémoire entre Unity et le DLL le nombre maximal de trackers est prédéfini, et si oui, nous enregistrons la nouvelle cible détectée.

3.2.3 Tracking de couleur

L'algorithme consiste à isoler une couleur d'une image et d'en suivre le déplacement. Pour cela, on passe en espace de couleur HSV, plus robuste que l'espace RGB quant aux changements de luminosité. Puis on seuille l'image obtenue pour obtenir une image binaire où les pixels blancs correspondent à la couleur souhaitée. On obtient ainsi une silhouette blanche sur fond noir. On utilise alors OpenCV pour déterminer les contours de la plus grande silhouette et obtenir ainsi une boîte englobante. De cette boîte, on peut extraire les informations de tracking qui nous intéressent, comme la position de l'objet dans l'image ou sa taille en pixel.

3.2.4 Approche hybride: tracking couleur et tracking OpenCV

Une fois que le tracking de couleur fonctionnel, nous avons voulu faire un tracking pouvant prendre en compte des objets quelconques. Par manque de temps, nous n'avons pu faire qu'une preuve de concept. En associant du tracking type détection et du tracking type tracker pur, nous avons pu grandement améliorer la précision et gérer les cas de perte de tracking, tout en gardant un coût raisonnable au niveau des performances.

Nous avons donc utilisé comme algorithme principal un tracker OpenCV qui permet de tracker des objets sans à priori sur ses informations, et en complément nous avons utiliser le tracker de couleur, qui repose sur de la détection, et qui se déclenche périodiquement. Grâce à ça nous avons pu tracker plus efficacement des objets de couleur, tout en préparant le workflow à de la détection de features (voir 3.4 pour les différents tracker testés).

3.3 Tests

Nous avons procédé à différents tests afin de vérifier la validité de nos algorithmes. Parmi eux se trouvent les tests de non-régression effectués via la console, simulant les différentes fonctionnalités du plugin utilisés sous Unity. On retrouve un simple test du DLL en effectuant l'initialisation et en le fermant ensuite, mais également un test de la caméra et de l'ouverture de vidéo. De plus, un test du workflow permet de vérifier le tracking en sélectionnant manuellement ce que l'on désire.

Nous avons implémenté ensuite un système d'annotation de vidéos afin de tester la validité de notre tracker sous Unity. L'idée est d'annoter manuellement une vidéo à un intervalle de frame régulier, et de récupérer la position, l'état, et la taille des objets dans un fichier xml. On va ensuite à nouveau parcourir la vidéo lors du tracking, et à chaque frame correspondante, les informations des différents objets suivis seront comparés.

D'autres tests ont été réalisés, notamment de performances. On peut utiliser le profiler de Unity pour avoir des détails plus précis sur les performances. Pour exposer des résultats et comparer dans le temps, un système de log permet de récupérer les résultats de ces performances a été établi. En plus de la vidéo et la webcam, nous avons aussi réalisé des tests virtuels simulant la caméra réelle, et permettant de contrôler chaque paramètre comme la distance et les paramètres intrinsèques notamment.

Enfin, nous faisons des tests qualitatifs à l'œil afin d'apporter un complément à l'ensemble des tests déjà effectués.

3.4 Résultats

3.4.1 Qualité

Afin d'évaluer qualitativement les différents algorithmes de tracking disponibles dans OpenCV, nous avons choisi de créer une scène utilisant une caméra virtuelle dans Unity3D où plusieurs cubes de couleurs différentes se déplacent dans et hors de la zone de détection. Nous avons également enregistré et annoté des vidéos en environnement réel mettant en scène différents cas d'utilisation, type d'objets et conditions d'éclairage. Pour être plus précis, nous avons testé les scénarios suivants : un unique carré en carton coloré qui se déplace dans et hors du champ de vision de la caméra, le même carré avec des déplacements rapides et brusques, 5 carrés de couleurs différentes en simultané, du texte imprimé, du texte manuscrit, une image imprimée, un scénario d'occultation où une partie du champ de vision de la caméra est bloquée par un obstacle, et enfin un scénario où les conditions d'éclairage alternent pendant toute la durée de l'enregistrement.

Grâce à une fonction de debug, il nous est possible de visualiser les bounding boxes détectées par les algorithmes de détection et tracking ainsi que de savoir si un objet a été perdu ou est encore suivi (exemple ci-dessous avec les figures 3 et 4). Cela nous permet d'évaluer à l'œil si un algorithme de tracking ne perd pas trop souvent les objets et à quel point il est précis ou a besoin d'être recentré grâce à l'algorithme de détection. De plus, dans le but d'affiner nos analyses, nous enregistrons dans des logs la différence entre les positions des objets/bounding boxes estimées par l'algorithme et les positions réelles des mêmes objets. Ces positions réelles correspondent aux coordonnées de l'objet dans la scène Unity3D dans le cadre des tests en virtuel, et aux annotations manuelles (toutes les 20 frames) des coordonnées de leur bounding box dans le cadre des vidéos réelles. Il est important de noter que la calibration de la caméra en réel n'a pas pu être effectuée, ce qui implique un biais sur les résultats obtenus. Comme il est identique pour chaque test enregistré, les chiffres qui en découlent ne sont pas à interpréter dans l'absolu mais relativement entre eux. Les unités ne seront pas indiquées pour cette même raison, bien qu'il s'agisse techniquement de mètres.

Les algorithmes testés sont Mosse-Color (MC) et CSRT-Color (CC). Le premier est plus rapide à l'exécution tandis que le second est plus robuste et plus précis de manière générale. Par ailleurs, CSRT appartient à la catégorie d'algorithmes de tracking d'OpenCV capable de changer dynamiquement la taille des bounding boxes pour un même objet afin de s'adapter à sa taille[ber]. Puisque la caméra est placée à la perpendiculaire de la zone de détection, la taille d'un objet peut apporter des informations sur sa position sur l'axe z de profondeur. Cette classe d'algorithmes est donc utile à l'estimation de position des objets. Nous avons également testé les algorithmes TLD, MEDIAN_FLOW et MIL qui sont également capables de redimensionner les bounding boxes. Cependant ceux-ci donnaient à vue d'œil trop de mauvais résultats et de faux positifs, c'est pourquoi ils ne sont pas inclus dans les graphiques présentés en annexe (voir figures 14 à 29).

Par souci de lisibilité, les graphiques en annexe ne comparent les algorithmes MOSSE-Color et CSRT-Color que pour un objet à la fois. Comme l'échelle n'est pas fixe il faut être prudent lorsqu'on compare les résultats entre 2 vidéos ou 2 objets d'une même vidéo. Une discontinuité indique une erreur de tracking c'est-à-dire en général que l'objet a été perdu alors qu'il n'aurait pas dû l'être. Un 0 indique que l'objet a été perdu comme ce qui était prévu.

L'analyse de ces graphiques d'erreur confirme les intuitions que nous avions au sujet des algorithmes MOSSE-Color et CSRT-Color. Que ce soit en virtuel ou en réel, CSRT est robuste et ne perd que rarement le suivi de l'objet. Ces résultats sont satisfaisant même lors d'occultations (figure 28), de sortie de la zone de détection (figure 23) ou de changements de luminosité (figure 29). Au contraire MOSSE perd souvent le tracking et doit attendre que l'algorithme de détection de couleur le recentre sur le bon objet pour en reprendre le suivi (figures 14 à 22). Il faut toutefois noter que MOSSE, lorsqu'il fonctionne correctement, obtient une erreur quasi identique à celle que produit CSRT. On remarque que les pertes de suivi de MOSSE ont surtout lieu pour les vidéos réelles, en particulier dans la vidéo où les mouvements de l'objet sont brusques (figure 24) et celle où les conditions d'éclairage changent régulièrement (figure 29). Il faut également noter que l'un comme l'autre ont des difficultés à tracker du texte, qu'il soit écrit (figure 25) ou manuscrit (figure **??**), ainsi que des images (figure 27). En effet la détection a tendance à remplacer le texte cible par un autre objet plus monochrome et entièrement coloré, comme les mains lorsqu'on manipule les cartes (voir figure 9 dans les screenshots ci-dessous). Il vaut mieux donc privilégier l'utilisation d'objets uniformément coloré comme les bouts de carton utilisés dans nos autres tests (figures 5 et 6).

On en conclut alors que CSRT est un algorithme de tracking plus fiable que MOSSE et serait mieux adapté au système de CARDS où il peut y avoir de nombreuses occultations, allées et venues dans la zone de détection et variations de luminosité, à condition que ses performances permettent de travailler en temps réel. Une autre remarque serait qu'il faudrait utiliser dans le cas de texte un algorithme de détection plus adapté que les couleurs, d'où les résultats obtenus.

Virtuel

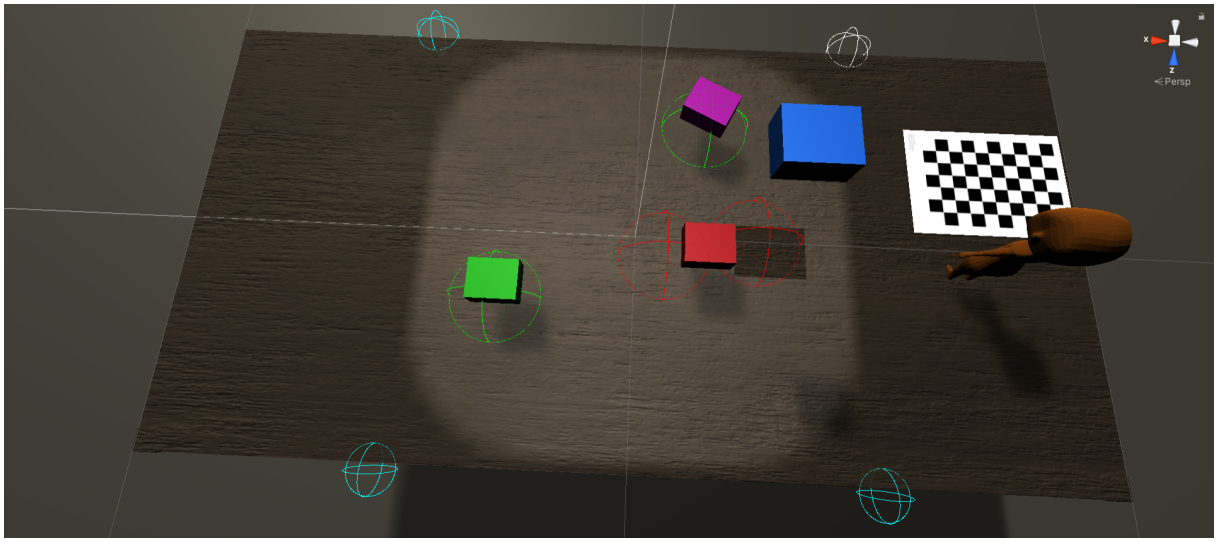

Fig. 3. MOSSE-Color utilisé dans la scène virtuelle, On perd le suivi lorsque les cubes bleu et rouge se superposent. (Les sphères bleues correspondent aux coins de la zone de détection, les vertes aux bounding boxes des objets trackés et les rouges aux bounding boxes des objets perdus.)

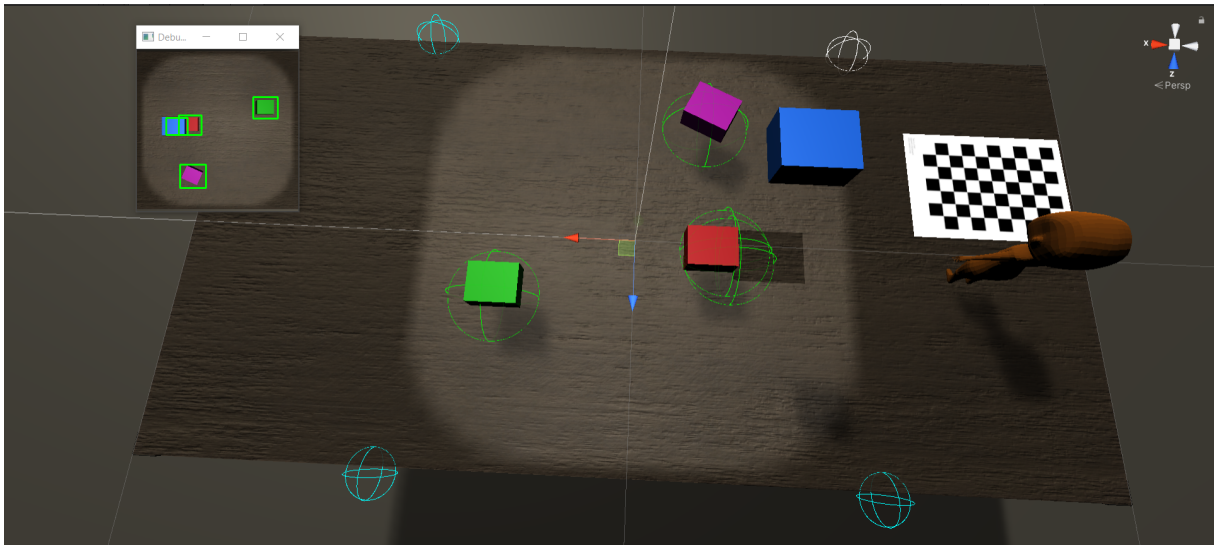

Fig. 4. CSRT-Color utilisé dans la scène virtuelle. On ne retrouve pas le problème précédent.

Objets multiples

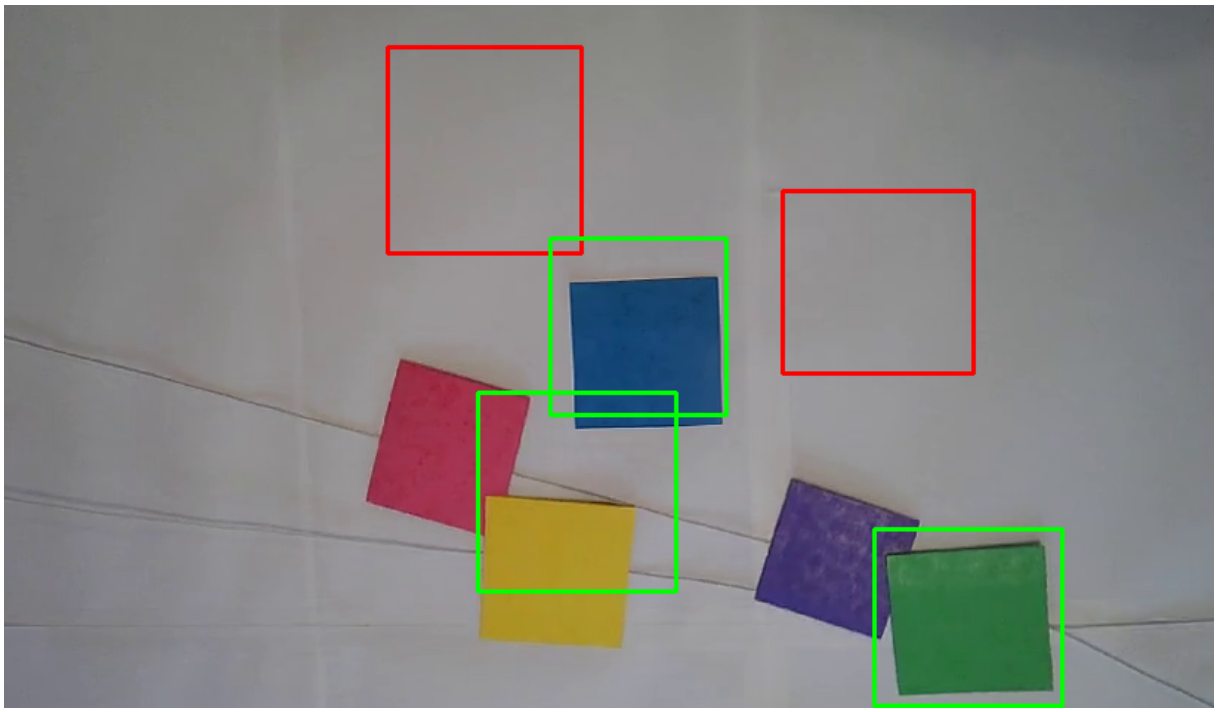

Fig. 5. MOSSE-Color utilisé dans la vidéo cibles multiples. Perte de tracking lors d'un mouvement brusque.

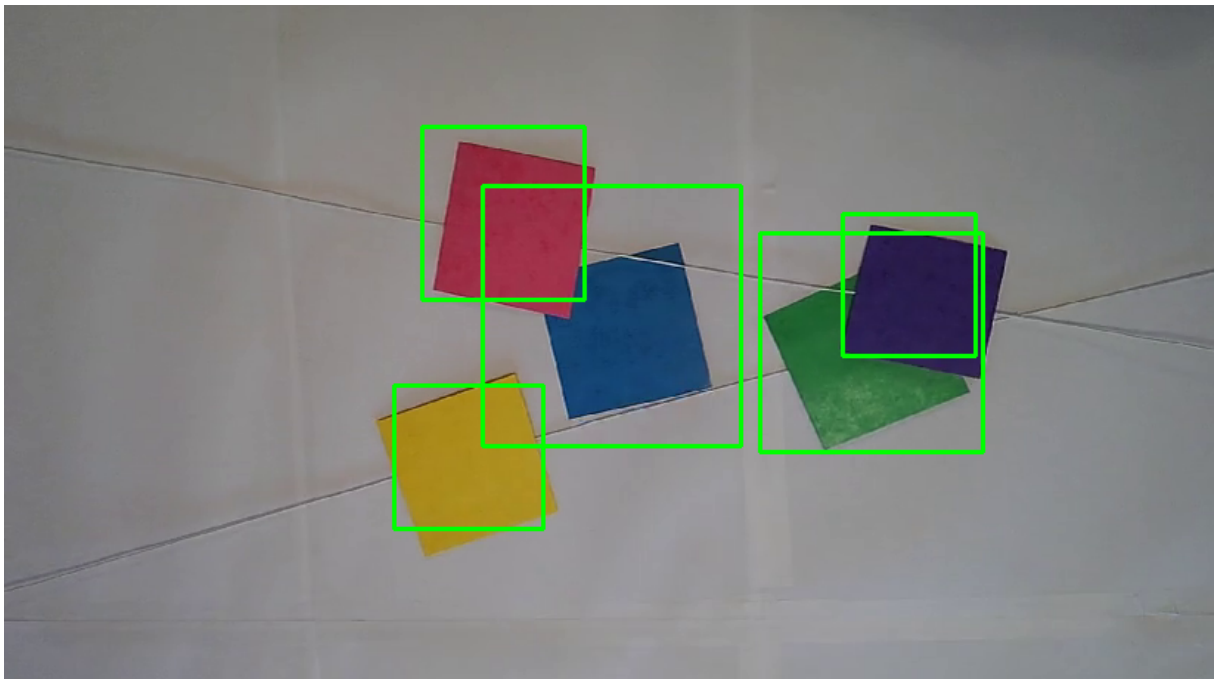

Fig. 6. CSRT-Color utilisé dans la vidéo cibles multiples. Pas de pertes de tracking lorsque des bounding boxes sont incluses dans d'autres.

Objets uniques

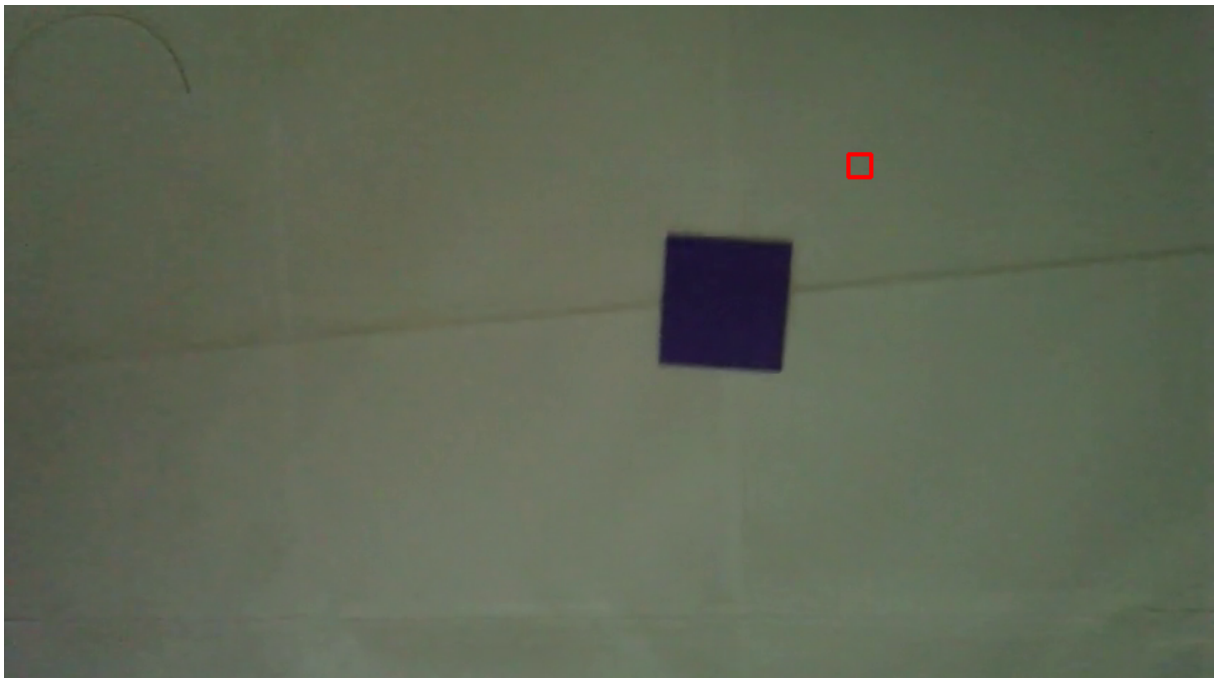

Fig. 7. MOSSE-Color dans la vidéo changements de luminosité. Perte de tracking lors du changement d'éclairage.

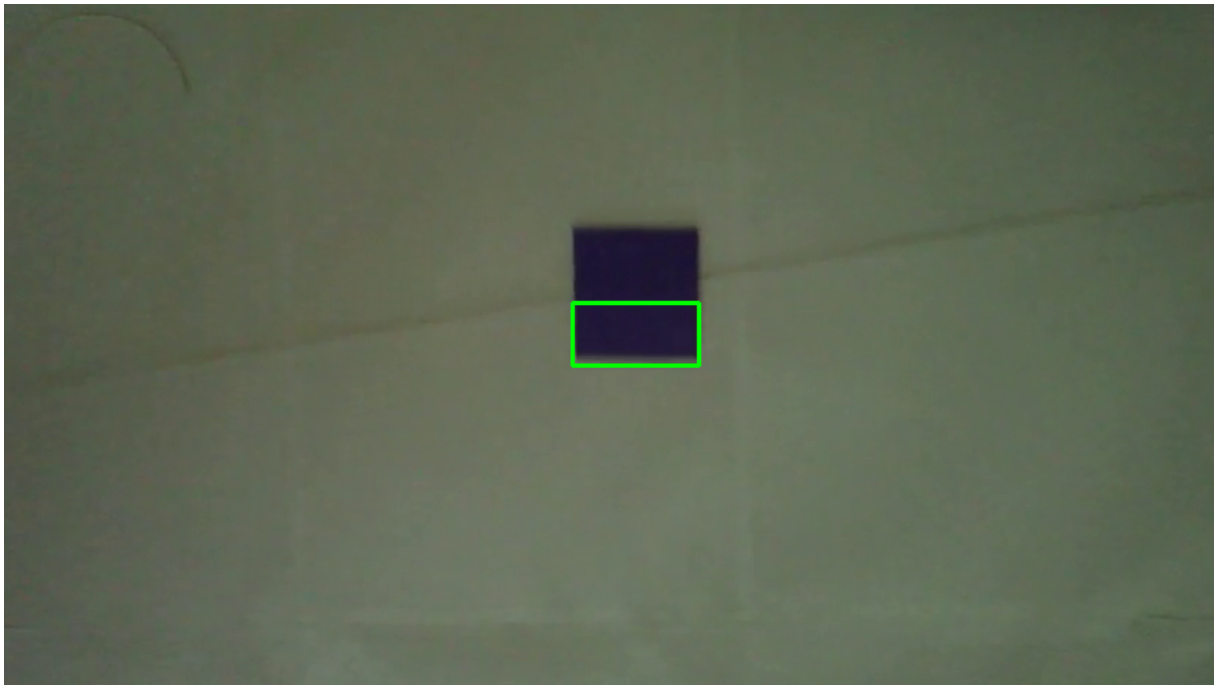

Fig. 8. CSRT-Color dans la vidéo changements de luminosité. On parvient à peu près à conserver une bounding box correcte.

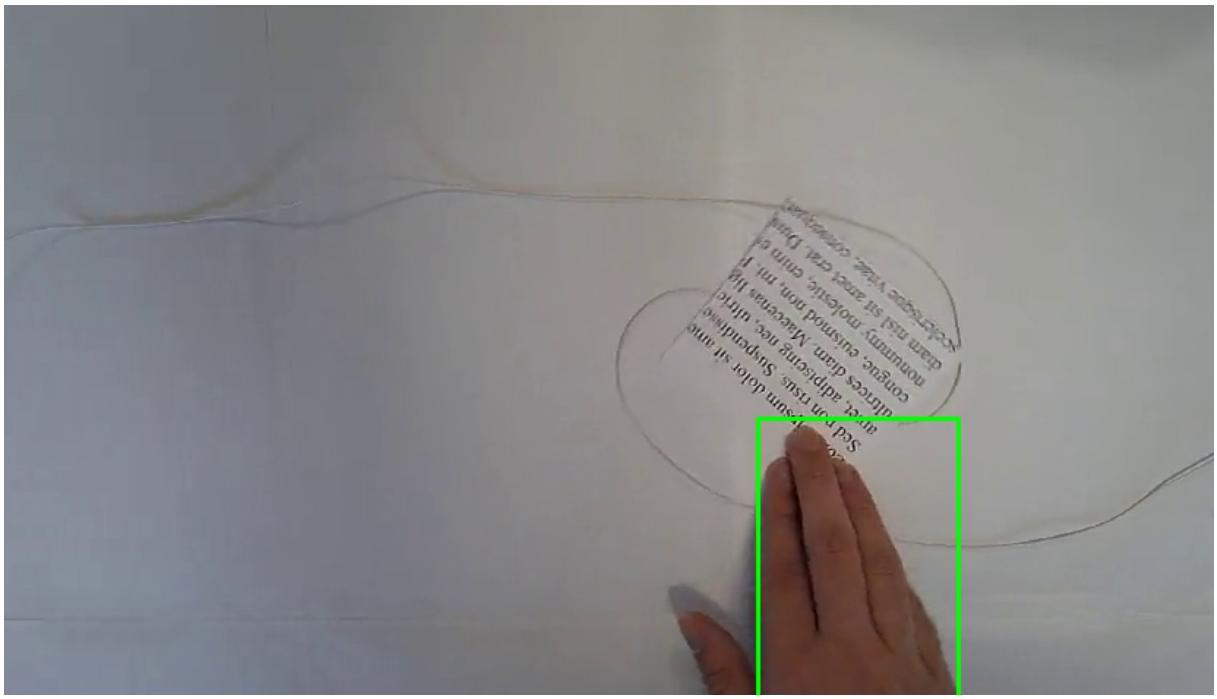

Fig. 9. CSRT-Color dans la vidéo texte imprimée. Le tracking du texte est rapidement perdu puis la main est détectée et trackée à sa place lorsqu'elle rentre ensuite dans la zone de détection.

3.4.2 Performances

Point de vue performances, on peut se concentrer sur trois critères : la source du média influant sur la taille de l'image en entrée, le nombre d'objets à tracker et enfin les algorithmes choisis. C'est pour cela que nous nous sommes concentrés sur la simulation et les vidéos de tracking pour un seul et plusieurs objets. Concernant les algorithmes, les meilleurs candidats étant Mosse-Color et CSRT-Color,nous nous sommes donc contenté de ces deux algorithmes.

Comme on peut le voir figure 10, Mosse-Color est largement plus performant que CSRT-Color. Nous pouvons juste observer que lorsque l'objet est perdu, on assiste à une hausse des performances du côté CSRT-Color avec un seul objet. Ce qui est en soit une confirmation de notre état de l'art [Mal], [Hon]. Nous pouvons par contre penser que puisqu'on ne peut pas toucher aux paramètres par défaut des trackers et que nous n'avons pas fait de multi-threading, il y aurait de la place pour des améliorations point de vue performances, mais pour l'instant on peut penser que CSRT-Color n'est pas viable pour du multitracking en temps réel.

On peut voir aussi des pics lors du Mosse-Color, et si on regarde plus en détail, d'après le profiler de Unity figure 11 en accord avec nos prévisions, il s'agit de l'algorithme utilisé dans **CheckTrack** qui est beaucoup plus lent que MOSSE. On peut voir cependant que cette alternance n'est pas visible pour CRST, car on peut supposer que les performances sont équivalentes entre cette détection de couleur et CRST. Notre algorithme ne permet donc de gagner en performances que si l'algorithme principal est assez performant. Pour gagner en performance, on peut soit allonger de taux de rafraîchissement du **CheckTrack**, soit utiliser des méthodes traditionnels pour améliorer les performances en traitement d'image (baisser la résolution, réduire le nombre d'opérations et de conversion, travailler sur moins d'objets, etc).

Fig. 10. Performances entre MOSSE-Color et CSRT-Color en échelle logarithmique.

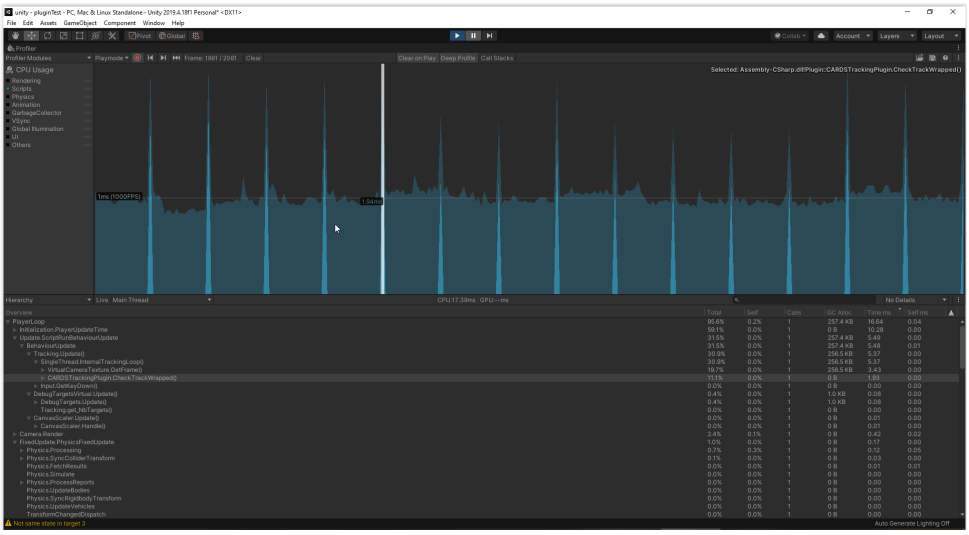

Fig. 11. Profiler Unity, checktrack et track MosseColor.

4 Conclusion

4.1 Gantt effectif vs prévisionnel

Nous avions initialement prévu un développement itératif, comme le montre le diagramme de Gantt prévisionnel (voir Annexes, Figure 12). Nous avions comme plan de développer un algorithme par semaine et utiliser la version la plus performante dans le projet final. Nous savions que cet objectif était pour le moins optimiste mais nous n'avions pas réalisé la charge de travail qu'allait représenter la création du projet de démonstrations sur Unity et l'installation de OpenCV.

Les principales différences avec le diagramme effectif (voir Annexes, Figure 13) viennent de l'abandon du tracking de features, des contours et de la gestion de l'occultation. Après discussion avec nos encadrants, selon les difficultés rencontrées et le temps imparti, nous avons décidé d'un commun accord de nous concentrer sur le tracking et la détection de couleur.

4.2 Améliorations possibles

Pour poursuivre ce projet, nous proposons plusieurs pistes d'amélioration. Elles permettent d'améliorer la précision du suivi d'objet, d'améliorer l'ergonomie du projet ou encore d'en améliorer les performances et la robustesse.

4.2.1 Amélioration de la précision

- **Calibration de caméra.** Pour ne pas perdre en précision, notamment sur les bords de l'image, il est possible d'utiliser des paramètres intrinsèques de la caméra. Ainsi, les distorsions causées par la caméra seraient atténuées et les images obtenues rendraient mieux compte des distances réelles. Cela améliorerait aussi la détection et le suivi lors d'un tracking de features.
- **Détection.** Lors du calcul de la MSE, le seuillage n'est pas normalisé à la taille de l'image. Ceci implique une sensibilité accrue à la luminosité et crée ainsi des problèmes de détection.
- **Vérification du tracking.** Lors du tracking, actuellement c'est la couleur qui est utilisée pour vérifier que l'on suit toujours le bon objet. Une bonne amélioration serait d'utiliser d'autres features, comme celles fournies par ORB[ERB11].
- **Deep learning.** Le tracker GOTURN[HTS16] d'OpenCV est une bonne opportunité d'utiliser le Deep Learning dans ce projet. Ce serait un bon moyen d'implémenter un tracker efficace sur un objet que l'on doit suivre dans plusieurs cas d'utilisation.
- **Estimation de position.** L'estimation des distances est aussi perfectible. La taille de départ pose problème et on ne gère pas encore la profondeur correctement, notamment sur la cohérence d'unités (mètre/pixels). En l'état la profondeur n'est pas utilisable.

4.2.2 Amélioration de l'ergonomie

- **Occultation.** La logique du projet Unity ne prend pas en compte quand les objets sont occultés ou s'ils sortent du champ de la caméra. Cela implique que les objets disparus restent suivis et pourraient impliquer une subsistance des objets virtuels concernés.
- **Personnalisation d'OpenCV.** On pourrait permettre à l'utilisateur de paramétrer lui même les tracker OpenCV ou le tracker de couleur. Cela nécessiterait soit un rebuild du projet Unity à chaque changement de paramètre mais pour un setup qui sera utilisé à moyen terme nous pensons que cela reste intéressant, et sinon un système de sérialisation des paramètres et de lecture depuis Unity serait l'idéal.
- **Tests.** La mise en place de tests plus complets et modulaires faciliterait le développement de nouvelles fonctionnalités. Par exemple, des utilitaires de benchmark pour les performances et la robustesse.
- **Unity.** Augmenter les scénarios d'utilisation dans Unity permettrait de mieux rendre compte des possibilités offertes par cette solution.

4.2.3 Amélioration des performances et de la robustesse

- **Thread.** Comme dans le projet CARDS original, l'utilisation de plusieurs threads permettrait d'améliorer les performances. Ce sera plus utile encore si on multiplie l'utilisation de plusieurs trackers en simultané, ou l'utilisation de machine learning.
- **Transition.** La transition entre les trackers (Track-CheckTrack) laisse encore à désirer et peut perdre les trackers. Fluidifier cette transition permettrait de rendre complètement invisible cette transition et améliorerait l'expérience utilisateur et la robustesse.
- **Gestion de mémoire.** La gestion de mémoire peut aussi être sujet à amélioration mais le fait d'utiliser des DLL C++ dans Unity rend cette gestion difficile. Les conversions entre les langages et les structures propres à chacun rend le risque de corruption de mémoire rée, et les statégies de gestion de mémoire sont nombreuses.

4.3 Bilan

Ce projet concernant la réalité augmentée spatiale, nous sommes satisfaits des résultats obtenus sans avoir pu accéder au matériel. Les résultats sur les objets virtuels sont bons et ceux sur les vidéos sont corrects. Il aurait toutefois été intéressant de pouvoir confronter les résultats obtenus à la réalité du terrain, notamment concernant les interactions homme-machine et les conditions de lumières et de contrastes d'une situation réelle.

Ce projet nous a particulièrement intéressé de part la diversité des approches mathématiques et algorithmiques mises en oeuvre dans les différentes solution envisagées. Le challenge de développement qu'a représenté la création de DLLs en plus d'un projet Unity a lui aussi été très formateur.

Nous tenons à remercier l'équipe Potioc de nous avoir permis de travailler sur ce projet, plus particulièrement Philippe Giraudeau et Joan Odicio Vilchez pour leur bienveillance et leurs conseils tout au long de ce projet. Merci également à Pascal Desbarats pour son encadrement dans la conduite de ce projet et à toute l'équipe enseignante du Master Informatique Image et Son de l'université de Bordeaux.

REFERENCES

[Bab08] Boris Babenko. Multiple Instance Learning: Algorithms and Applications. 2008.

[ber] berak. **Scale-adaptive object tracking**. https://answers.opencv.org/question/233484/scale-adaptive-object-tracking/. Dernier accès 22/03/2021.

[BGO+16] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. Simple Online and Realtime Tracking. *arXiv e-prints*, page arXiv:1602.00763, February 2016.

[BWL20] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv e-prints*, page arXiv:2004.10934, April 2020.

[Coh] Jeremy Cohen. **Tracking par Computer Vision**. https://medium.com/france-school-of-ai/tracking-par-computer-vision-90e5111cbb86. Dernier accès 16/03/2021.

[ERB11] Vincent Rabaud Kurt Konolige Ethan Rublee and Gary Bradski. Orb: an efficient alternative to sift or surf. In *International Conference on Computer Vision*. IEEE, 2011.

[GDDM13] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv e-prints*, page arXiv:1311.2524, November 2013.

[Gir15] Ross Girshick. Fast R-CNN. *arXiv e-prints*, page arXiv:1504.08083, April 2015.

[GOSR+19] Philippe Giraudeau, Alexis Olry, Joan Sol Roo, Stéphanie Fleck, David Bertolo, Robin Vivian, and Martin Hachet. **CARDS: A Mixed-Reality System for Collaborative Learning at School**. In *ACM ISS'19 - ACM International Conference on Interactive Surfaces and Spaces*, ISS '19: Proceedings of the 2019 ACM International Conference on Interactive Surfaces and Spaces, pages 55–64, Daejeon, South Korea, November 2019.

[Hon] Taeha Hong. **Test Object Tracking using OpenCV**. https://medium.com/@xogk39/test-object-tracking-using-opencv-58066d490d1e. Dernier accès 20/03/2021.

[HTS16] David Held, Sebastian Thrun, and Silvio Savarese. Learning to Track at 100 FPS with Deep Regression Networks. *arXiv e-prints*, page arXiv:1604.01802, April 2016.

[JSBH18] Ilchae Jung, Jeany Son, Mooyeol Baek, and Bohyung Han. Real-Time MDNet. *arXiv e-prints*, page arXiv:1808.08834, August 2018.

[JSS18] Nelson Monzón Javier Sánchez and Agustín Salgado. An Analysis and Implementation of the Harris Corner Detector. *IPOL Journal : Image Processing On Line*, October 2018.

[KMM12] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 34(7):1409–1422, July 2012.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[KY55] H. W. Kuhn and Bryn Yaw. The hungarian method for the assignment problem. *Naval Res. Logist. Quart*, pages 83–97, 1955.

[Luc] Walter Lucetti. **[Tutorial OpenCV] "Ball Tracker" using Kalman filter**. https://www.myzhar.com/blog/tutorials/tutorial-opencv-ball-tracker-using-kalman-filter/. Dernier accès 16/03/2021.

[Mal] Satya Mallick. **Object Tracking using OpenCV (C++/Python)**. https://learnopencv.com/object-tracking-using-opencv-cpp-python/. Dernier accès 20/03/2021.

[MB17] Darshana Mistry and Asim Banerjee. Comparison of Feature Detection and Matching Approaches: SIFT and SURF. *GRD Journals- Global Research and Development Journal for Engineering*, March 2017.

[MCF10] Christoph Strecha Michael Calonder, Vincent Lepetit and Pascal Fua. BRIEF: Binary Robust Independent Elementary Features. *European Conference on Computer Vision*, 2010.

[NH15] Hyeonseob Nam and Bohyung Han. Learning Multi-Domain Convolutional Neural Networks for Visual Tracking. *arXiv e-prints*, page arXiv:1510.07945, October 2015.

[NZH+16] Guanghan Ning, Zhi Zhang, Chen Huang, Zhihai He, Xiaobo Ren, and Haohong Wang. Spatially Supervised Recurrent Convolutional Neural Networks for Visual Object Tracking. *arXiv e-prints*, page arXiv:1607.05781, July 2016.

[Opea] OpenCV. **AruCo Library**. https://sourceforge.net/projects/aruco/files/3.1.2/aruco-3.1.2.zip/download. Dernier accès 20/03/2021.

[Opeb] OpenCV. **Camshift & Meanshift tutorial**. https://docs.opencv.org/master/d7/d00/tutorial_meanshift.html. Dernier accès 16/03/2021.

[OR15] Edouard Oyallon and Julien Rabin. An Analysis of the SURF Method. *IPOL : Image Processing On Line*, July 2015.

[PJS16] Tomas Dulik1 Peter Janku1, Karel Koplik and Istvan Szabo. Comparison of tracking algorithms implemented in OpenCV. *MATEC Web of Conferences*, 2016.

[RDGF15] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *arXiv e-prints*, page arXiv:1506.02640, June 2015.

[RHGS15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *arXiv e-prints*, page arXiv:1506.01497, June 2015.

[Teca] Unity Technologies. **Native plugins - Adding Love to an API (or How to Expose C++ in Unity)** . https://fr.slideshare.net/unity3d/adding-love-to-an-api-or-how-to-expose-c-in-unity. Dernier accès 20/03/2021.

[Tecb] Unity Technologies. **Unity3D**. https://unity.com/fr. Dernier accès 20/03/2021.

[WBP17] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. Simple Online and Realtime Tracking with a Deep Association Metric. *arXiv e-prints*, page arXiv:1703.07402, March 2017.

[YF99] Robert E. Schapire Yoav Freund. A Short Introduction to Boosting. *Journal of Japanese Society for Artificial Intelligence*, 1999.

[ZK10] Jiri Matas Zdenek Kalal, Krystian Mikolajczyk. Forward-backward error: Automatic detection of tracking failures. *Pattern Recognition (ICPR), 2010 20th International Conference on*, 2010.

5 ANNEXES

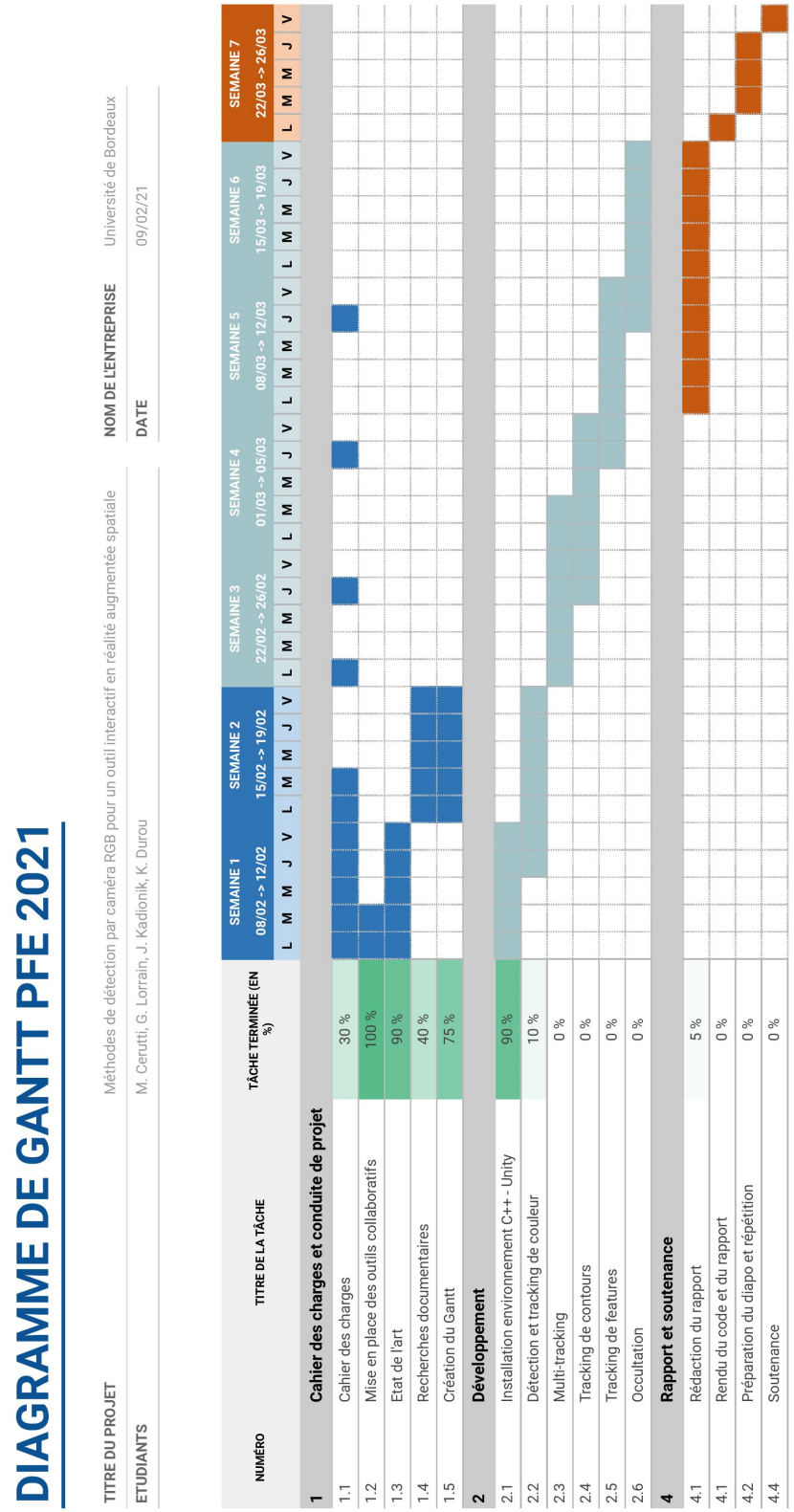

Fig. 12. Diagramme de Gantt prévisionnel

Fig. 13. Diagramme de Gantt effectif

DIAGRAMME DE GANTT PFE 2021

TITRE DU PROJET	Méthodes de détection par caméra RGB pour un outil interactif en réalité augmentée spatiale	NOM DE L'ENTREPRISE	Université de Bordeaux
ETUDIANTS	M. Cerutti, G. Lorrain, J. Kadionik, K. Durou	DATE	17/03/21

NUMÉRO	TITRE DE LA TÂCHE	TÂCHE TERMINÉE (EN %)	SEMAINE 1 08/02 -> 12/02					SEMAINE 2 15/02 -> 19/02					SEMAINE 3 22/02 -> 26/02					SEMAINE 4 01/03 -> 05/03					SEMAINE 5 08/03 -> 12/03					SEMAINE 6 15/03 -> 19/03					SEMAINE 7 22/03 -> 26/03				
			L	M	M	J	V	L	M	M	J	V	L	M	M	J	V	L	M	M	J	V	L	M	M	J	V	L	M	M	J	V	L	M	M	J	V
1	**Cahier des charges et conduite de projet**																																					
1.1	Release note	100 %																																				
1.2	Mise en place des outils collaboratifs	100 %																																				
1.3	Etat de l'art	100 %																																				
1.4	Recherches documentaires	100 %																																				
1.5	Création du Gantt	100 %																																				
2	**Développement**																																					
2.1	Installation environnement C++ - Unity	100 %																																				
2.2	Détection et tracking de couleur	70 %																																				
2.3	Multi-tracking	100 %																																				
2.4	Annotation de vidéos	100 %																																				
2.5	Estimation de position	50 %																																				
2.6	Calibration de la caméra	80 %																																				
2.5	Tracking de contours	0 %																																				
2.6	Tracking de features	0 %																																				
2.7	Occultation	20 %																																				
4	**Rapport et soutenance**																																					
4.1	Rédaction du rapport	5 %																																				
4.1	Rendu du code et du rapport	0 %																																				
4.2	Préparation du diapo et répétition	0 %																																				
4.4	Soutenance	0 %																																				

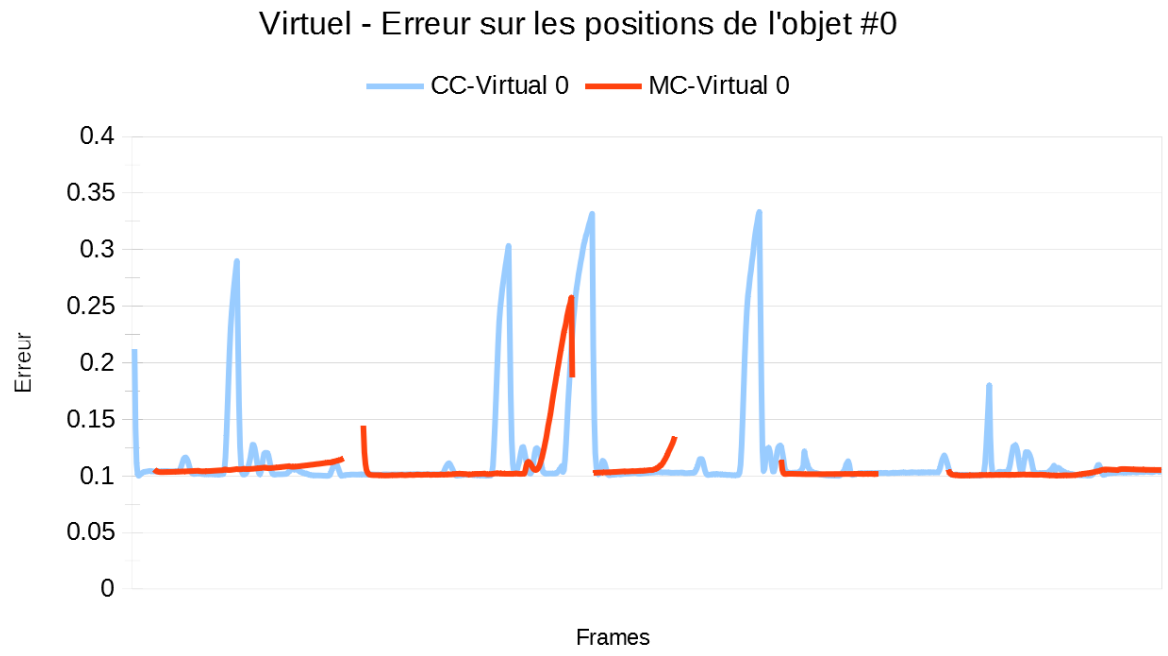

Fig. 14. Erreur sur la position de l'objet 0 en virtuel entre Mosse-Color et CSRT-Color.

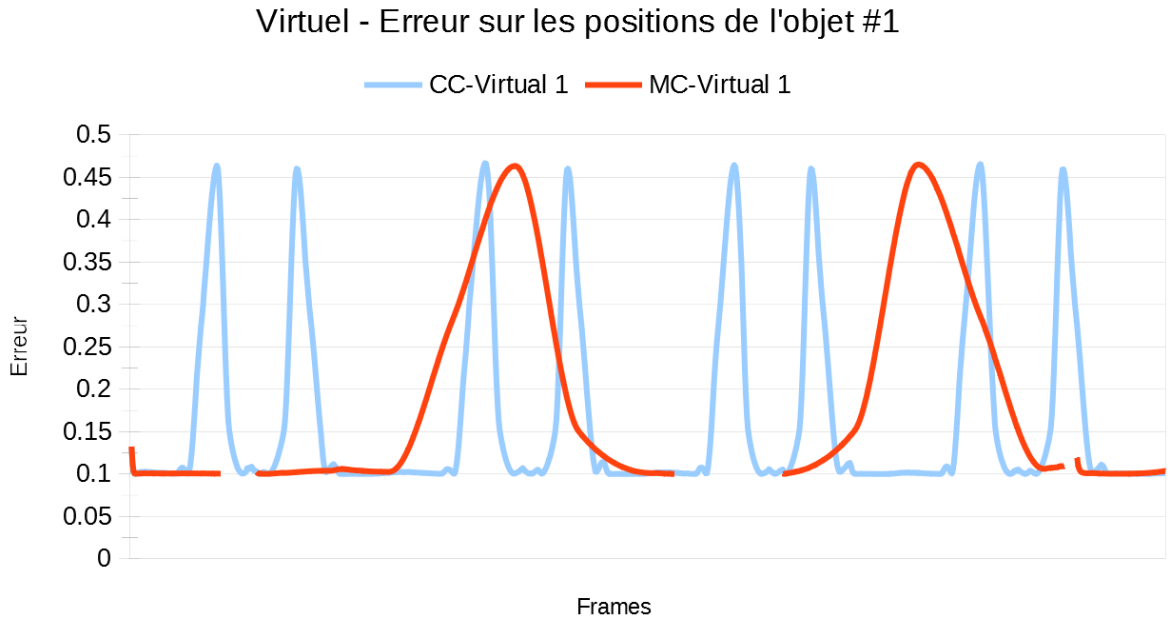

Fig. 15. Erreur sur la position de l'objet 1 en virtuel entre Mosse-Color et CSRT-Color.

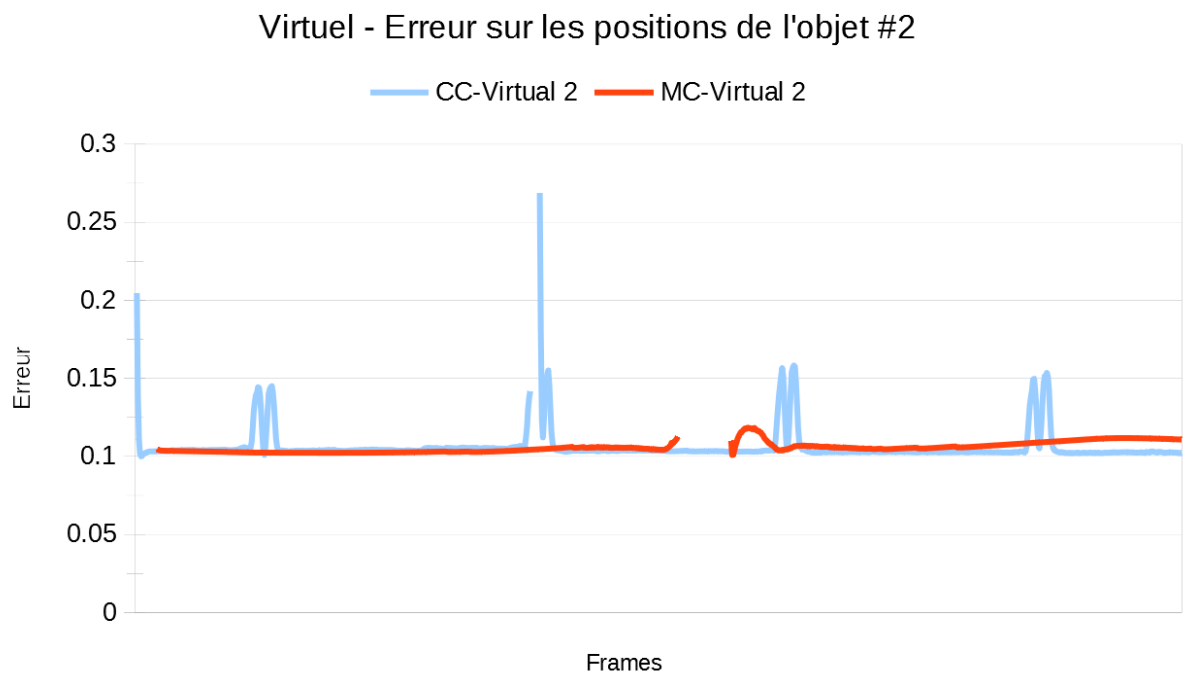

Fig. 16. Erreur sur la position de l'objet 2 en virtuel entre Mosse-Color et CSRT-Color.

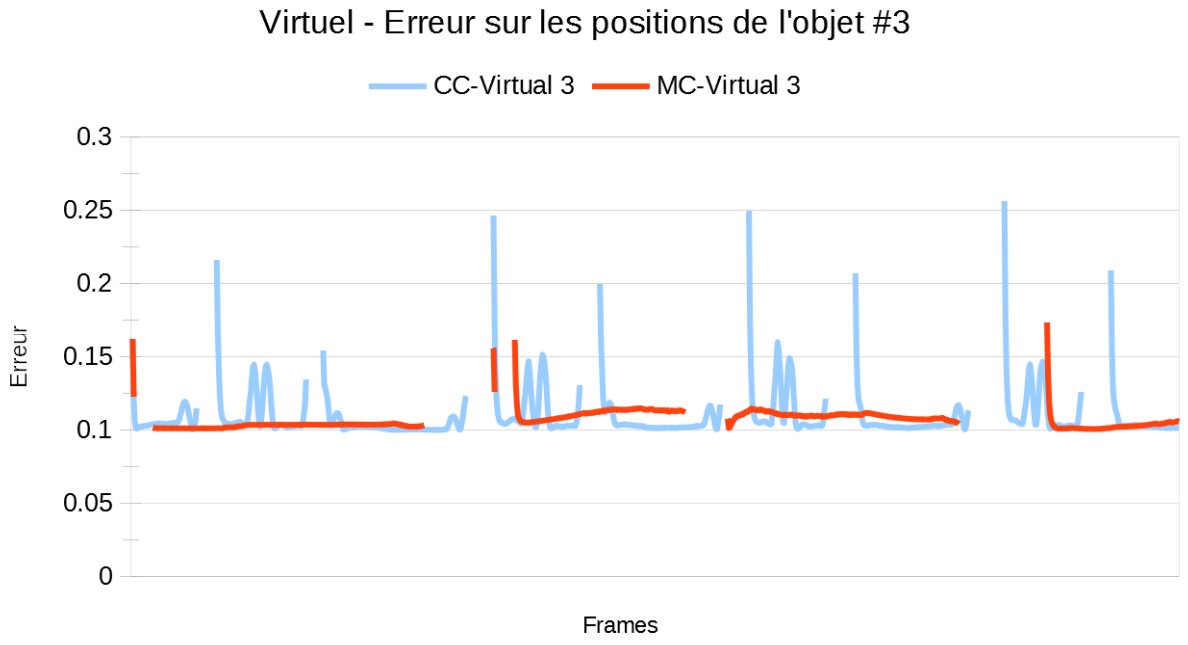

Fig. 17. Erreur sur la position de l'objet 3 en virtuel entre Mosse-Color et CSRT-Color.

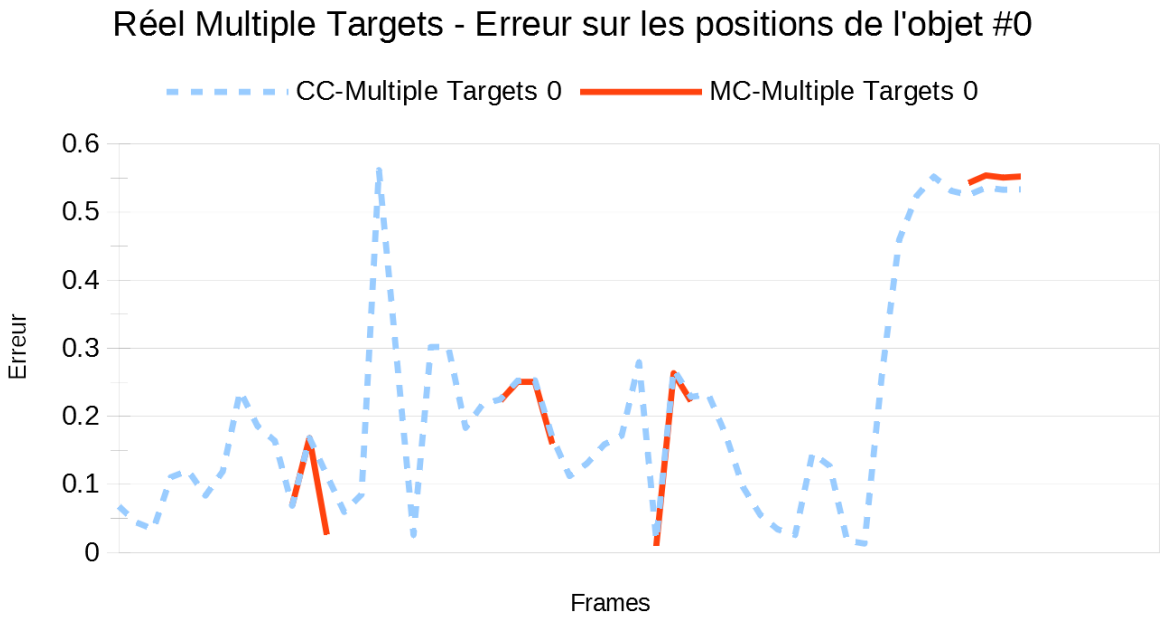

Fig. 18. Erreur sur la position de l'objet 0 en réel entre Mosse-Color et CSRT-Color.

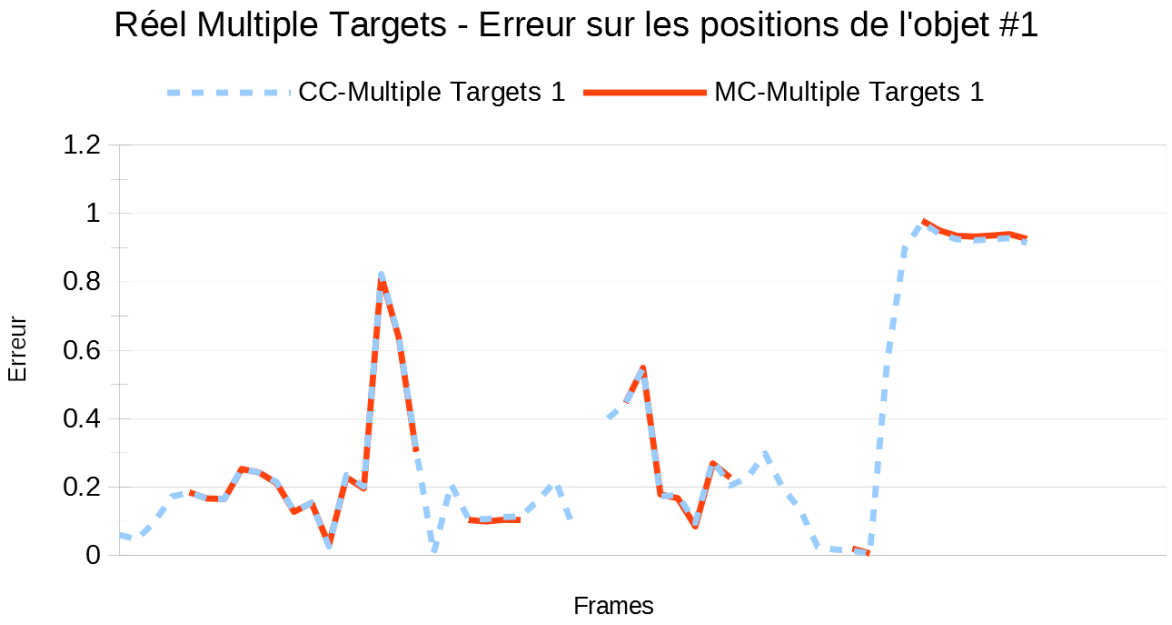

Fig. 19. Erreur sur la position de l'objet 1 en réel entre Mosse-Color et CSRT-Color.

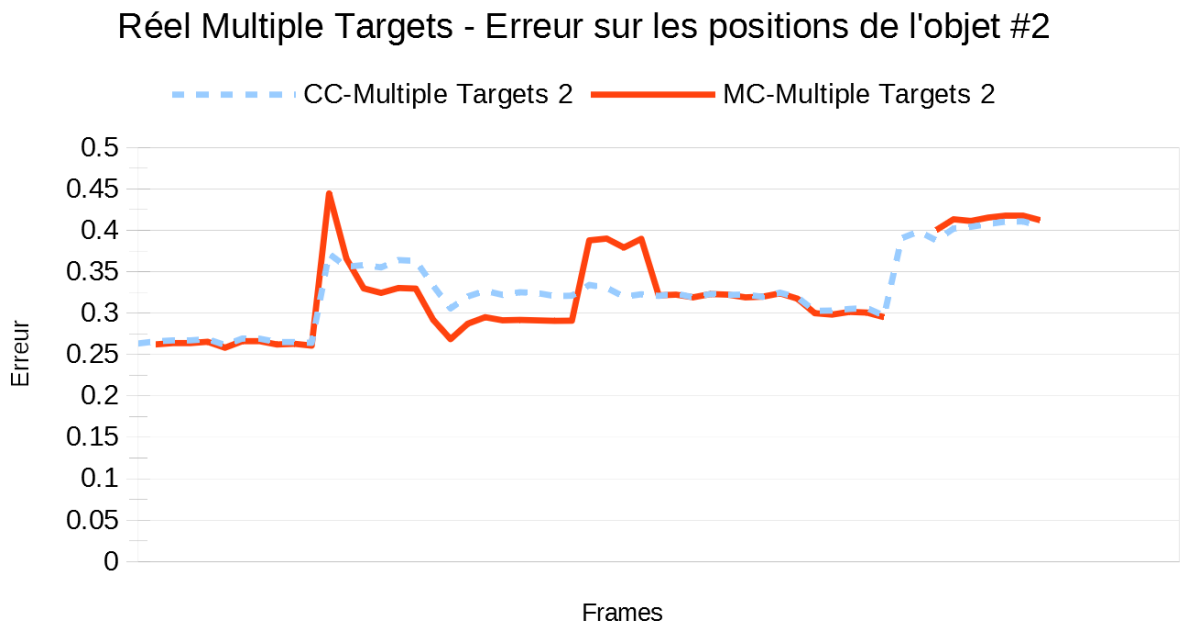

Fig. 20. Erreur sur la position de l'objet 2 en réel entre Mosse-Color et CSRT-Color.

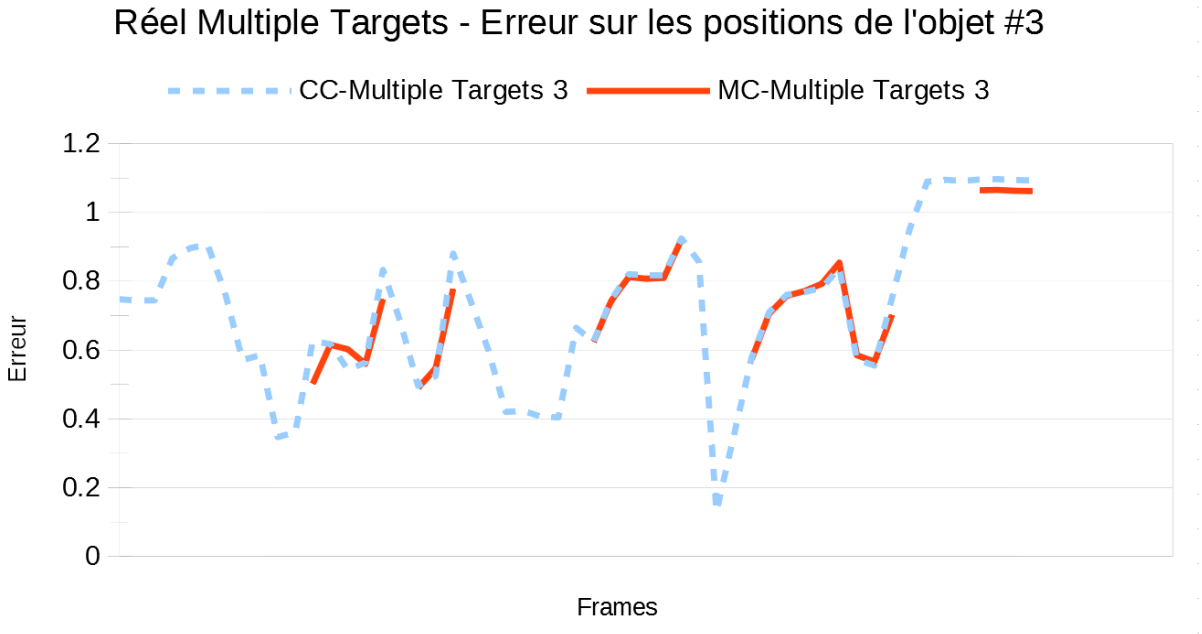

Fig. 21. Erreur sur la position de l'objet 3 en réel entre Mosse-Color et CSRT-Color.

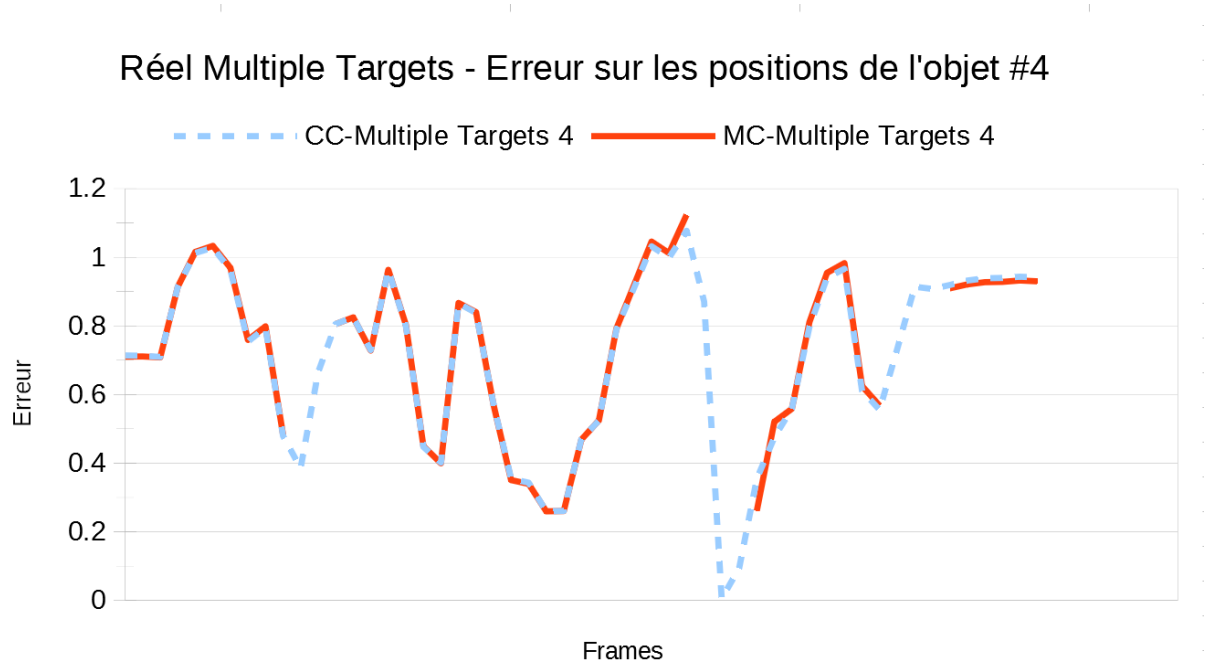

Fig. 22. Erreur sur la position de l'objet 4 en réel entre Mosse-Color et CSRT-Color.

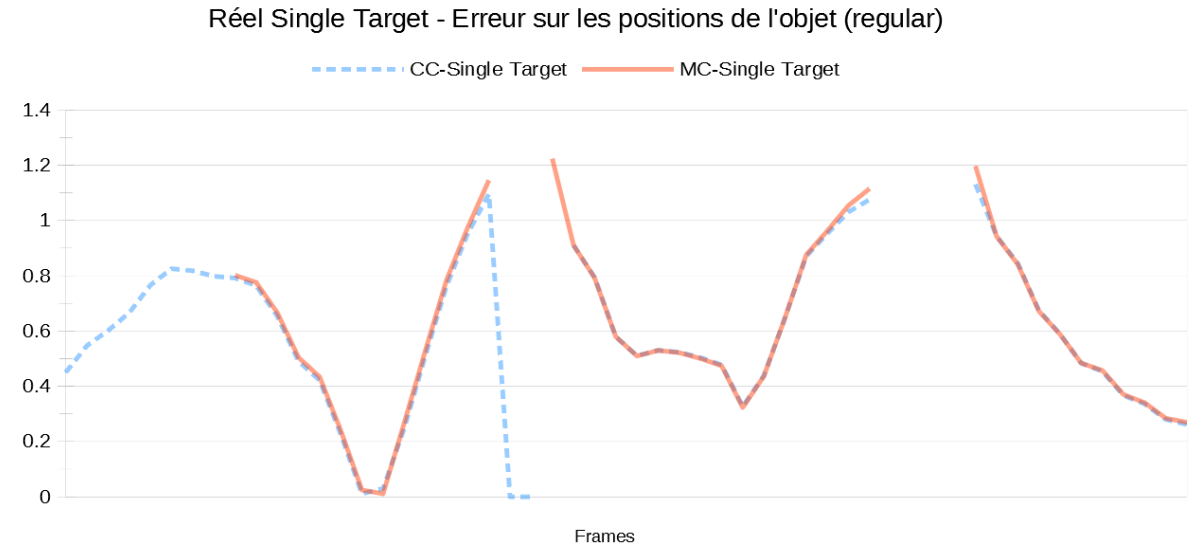

Fig. 23. Erreur sur la position de l'objet en réel entre Mosse-Color et CSRT-Color (vidéo objet unique).

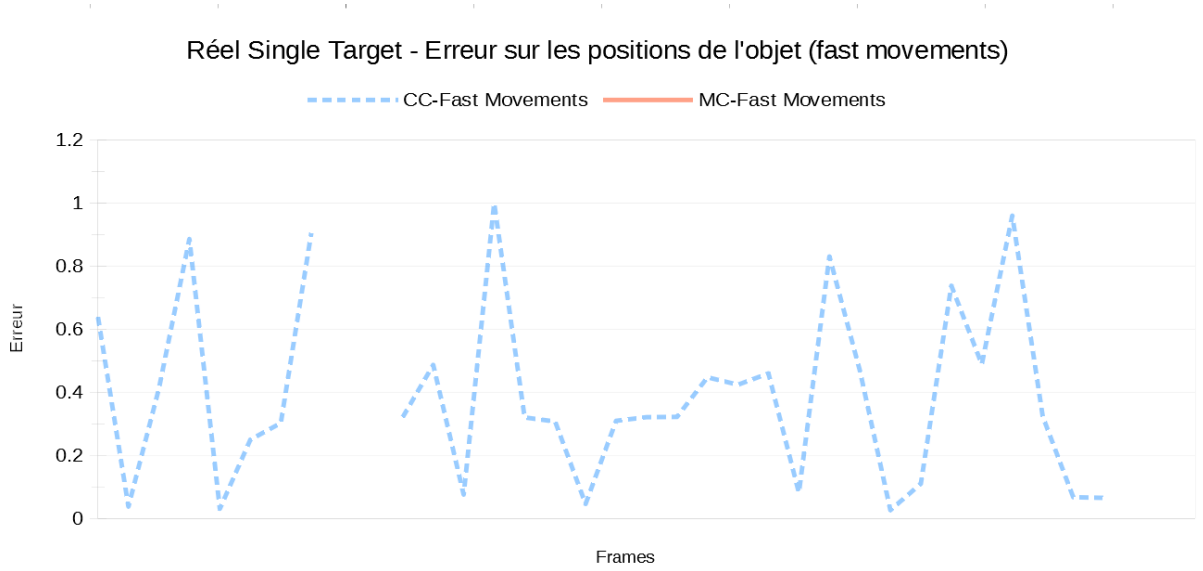

Fig. 24. Erreur sur la position de l'objet en réel entre Mosse-Color et CSRT-Color (vidéo déplacements rapides).

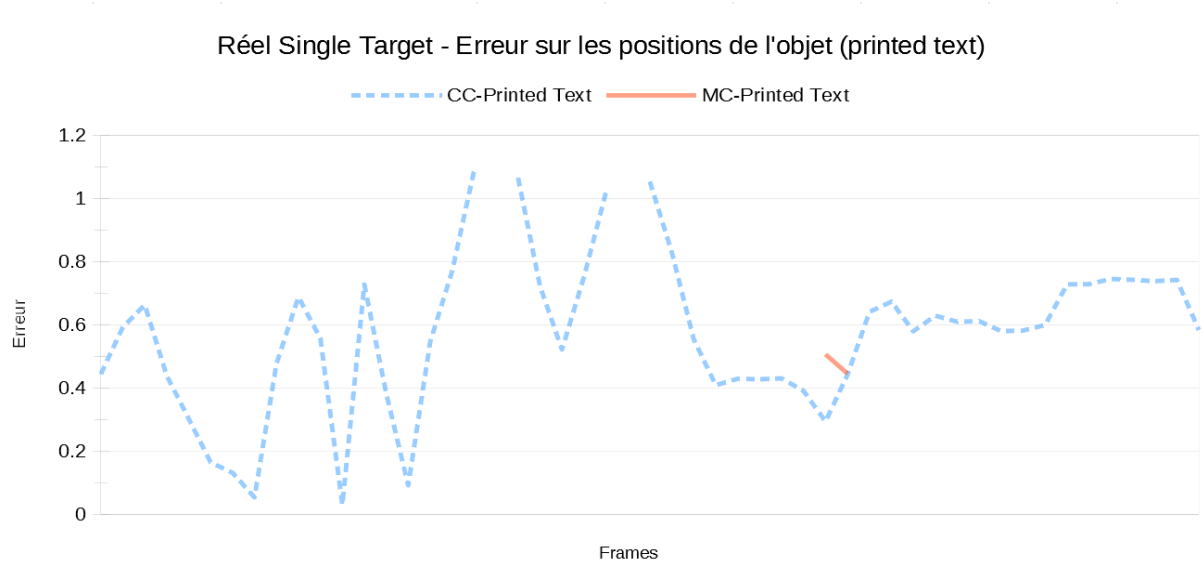

Fig. 25. Erreur sur la position de l'objet en réel entre Mosse-Color et CSRT-Color (vidéo texte imprimé).

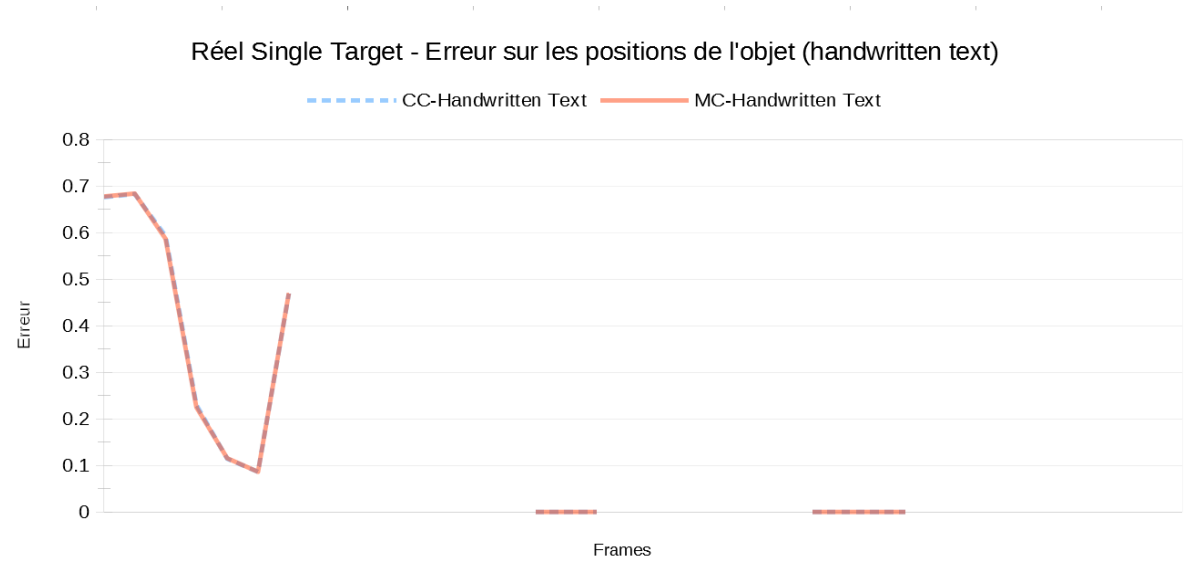

Fig. 26. Erreur sur la position de l'objet en réel entre Mosse-Color et CSRT-Color (vidéo texte manuscrit).

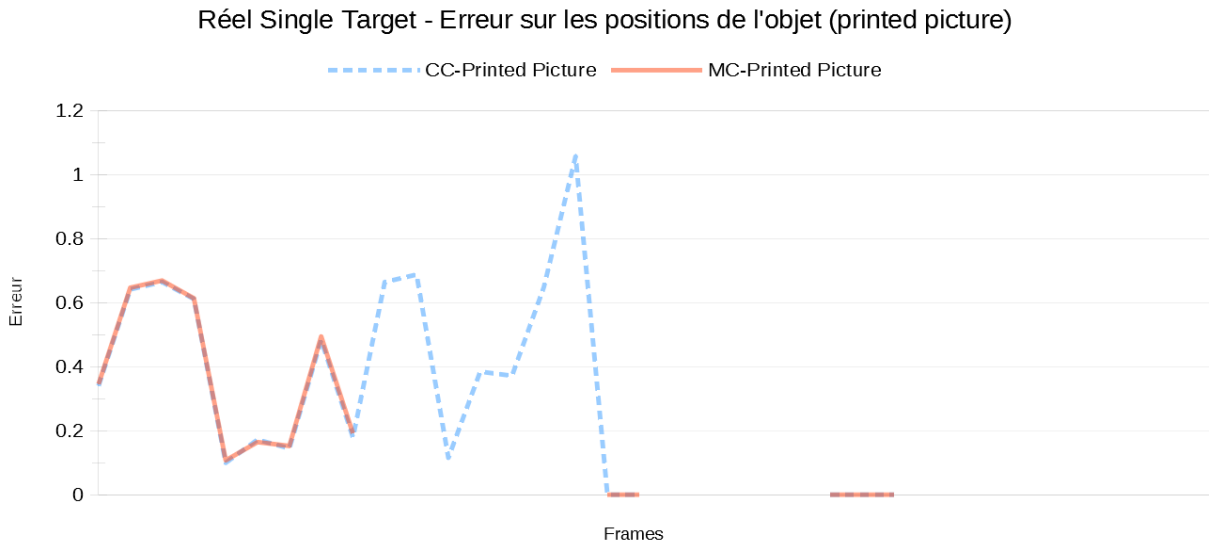

Fig. 27. Erreur sur la position de l'objet en réel entre Mosse-Color et CSRT-Color (vidéo image imprimée).

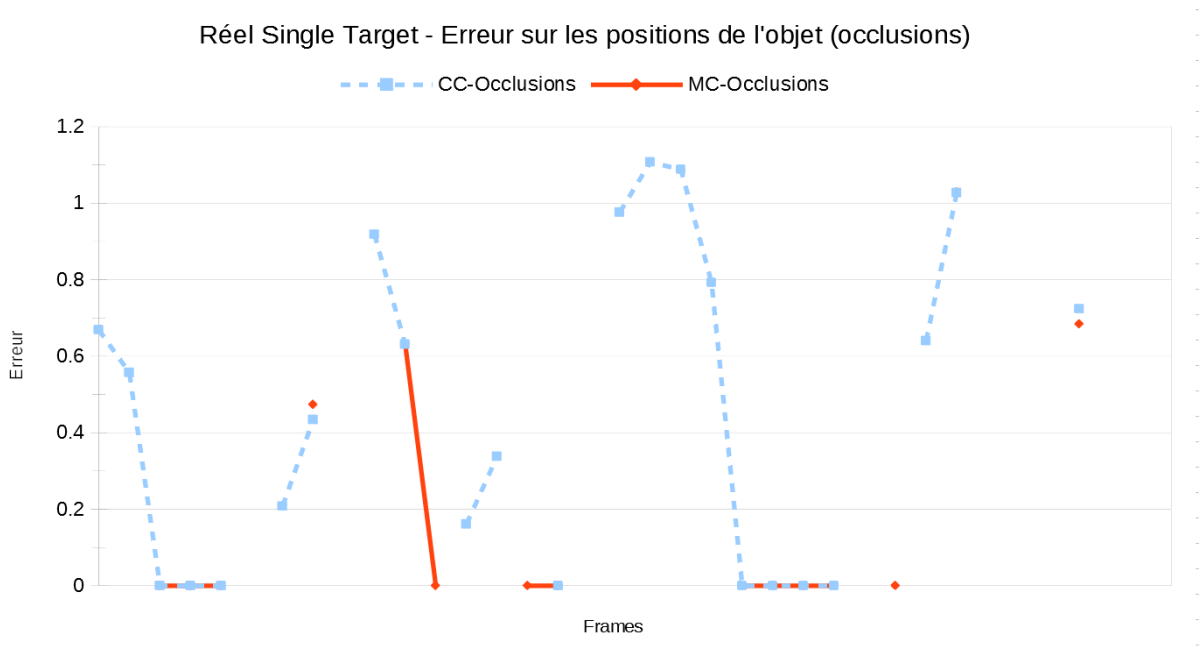

Fig. 28. Erreur sur la position de l'objet en réel entre Mosse-Color et CSRT-Color (vidéo occultations).

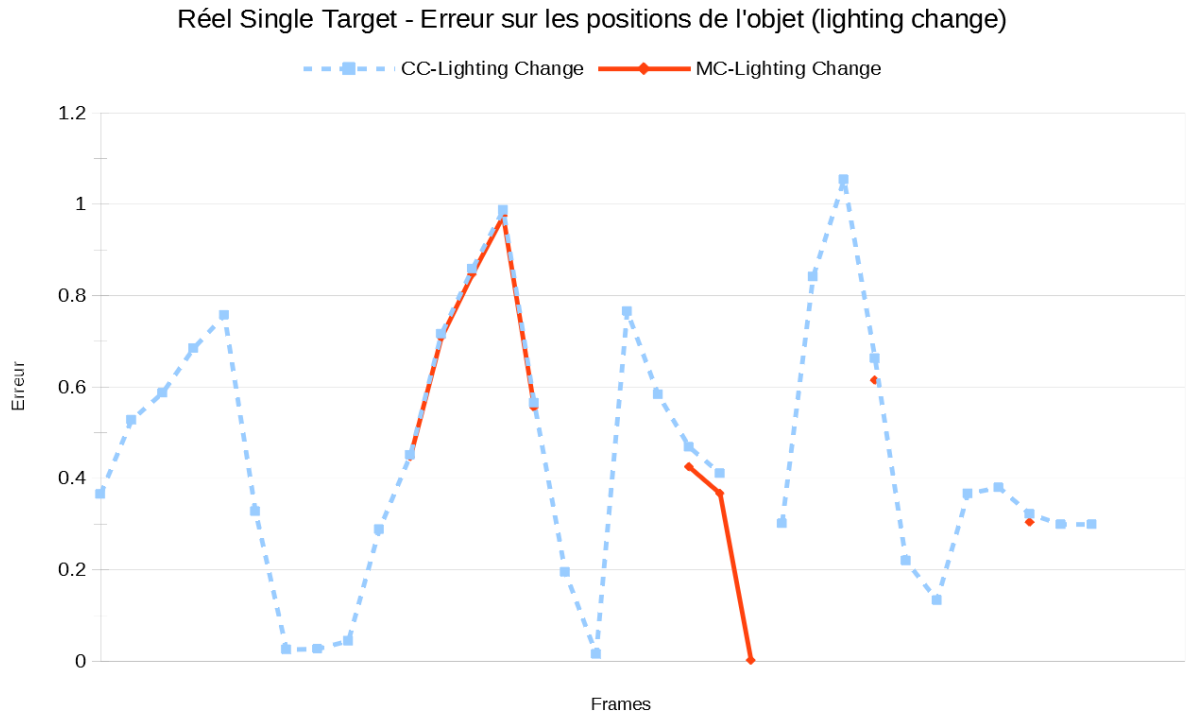

Fig. 29. Erreur sur la position de l'objet en réel entre Mosse-Color et CSRT-Color (vidéo changements de conditions d'éclairage).