Rapport de capitalisationStage de synthèse en robotique

Stagiaire : Votre Nom
Encadrant : Nom de l'encadrant
Période : Janvier 2026

Introduction

Ce rapport de capitalisation synthétise l'ensemble des connaissances et des échanges réalisés pendant un stage visant à développer un module d'évitement d'obstacles dynamique pour un robot mobile équipé d'une caméra événementielle (DAVIS 346) et fonctionnant sous ROS 2. Les conversations avec des intelligences artificielles ont permis d'obtenir des explications, des conseils pratiques et des plans d'implémentation. Le présent document regroupe ces informations en plusieurs chapitres pour en faire un ouvrage de référence. Il commence par rappeler le contexte du stage et les objectifs visés, présente les principes fondamentaux des capteurs et des algorithmes employés, puis détaille la migration logicielle vers ROS 2, la programmation C++ moderne, l'intégration sur le robot Limo et les bonnes pratiques découvertes au fil des échanges.

Contexte du stage et objectifs

Le stage s'inscrit dans un projet de robotique mobile où l'objectif est de détecter et d'éviter des obstacles dynamiques en exploitant une caméra événementielle de type DAVIS 346. Ce capteur fournit simultanément des flux d'événements (changement de luminosité par pixel) et des images standard (APS), ce qui permet d'obtenir une grande précision temporelle tout en restant compatible avec les algorithmes de vision traditionnels. La mission consistait à :

- Comprendre l'article de l'ICRA 2023 intitulé *Event-based Real-time Moving Object Detection Based on IMU Ego-motion Compensation*. Cet article propose une méthode de compensation de mouvement non linéaire basée sur les données d'une centrale inertielle (IMU) pour isoler les objets mobiles dans un flux d'événements. L'approche est validée en temps réel avec une latence inférieure à 10 ms, ce qui est essentiel pour les drones ou les robots rapides.
- Porter le code existant de ROS 1 vers ROS 2 (Humble). Le projet rpg_dvs_ros fournit un exemple de compensation de mouvement en ROS 1, mais il n'est pas complet et n'intègre pas la segmentation ni le regroupement d'objets. Le passage à ROS 2 implique de revoir la structure des nœuds, la gestion des paramètres, le système de build (ament_cmake) et les politiques *Quality of Service* (QoS).
- Implémenter les étapes manquantes de la chaîne de traitement de l'article : génération des images temporelles (*time image*), seuillage dynamique, algorithme de clustering

DBSCAN, et, à terme, stratégie d'évitement d'obstacles.

- Configurer un environnement de développement efficace (VS Code, colcon, compile_commands) et assurer le bon fonctionnement du driver de la caméra (libcaer_driver) ainsi que des messages associés (event_camera_msgs et dvs_msgs).

Au fil des échanges, de nombreux concepts transverses ont été abordés : principes des capteurs événementiels, bases et subtilités du langage C++, conception orientée objet et RAII, multithreading, configuration de ROS 2 (QoS, namespaces, remappings), ainsi que les bonnes pratiques pour le développement et le débogage.

Principes des caméras événementielles

Les caméras événementielles représentent une rupture par rapport aux capteurs d'images classiques. Elles s'inspirent du fonctionnement biologique des yeux et détectent uniquement les changements de luminosité. Chaque pixel travaille indépendamment et produit un *événement* lorsqu'il constate un changement d'intensité. Ce fonctionnement asynchrone permet d'atteindre une résolution temporelle de l'ordre de la microseconde et une dynamique de contraste supérieure à 120 dB, supprimant presque totalement le flou de mouvement 【86159226765082†L129-L158】. Un capteur DAVIS combine un détecteur d'événements (DVS) et une matrice APS capable de fournir des images classiques. Les flux d'événements sont donc complétés par des images basse fréquence utiles pour la calibration ou l'affichage 【86159226765082†L198-L200】.

Définition d'un événement

Un événement est défini par un quadruplet (t, x, y, p) où

- t est l'horodatage précis de l'événement;
- (x, y) sont les coordonnées du pixel concerné;
- $p \in \{+1, -1\}$ indique la polarité (augmentation ou diminution de luminosité).

Contrairement à un flux vidéo à 30 images par seconde, un capteur événementiel génère un nombre variable d'événements proportionnel aux changements dans la scène. Cela permet de traiter les mouvements rapides tout en réduisant la quantité de données à transmettre.

Avantages et limites

Les principaux avantages des caméras événementielles sont :

- **Résolution temporelle élevée** : les pixels réagissent immédiatement aux changements, avec des latences de quelques microsecondes 【86159226765082†L129-L158】. Cela permet de suivre des objets rapides sans flou.

- **Plage dynamique étendue** : la dynamique de 120 dB dépasse de loin celle des caméras classiques, ce qui permet de fonctionner dans des scènes à très fort contraste 【86159226765082†L129-L158】.
- **Réduction de la bande passante** : seuls les changements sont transmis, ce qui diminue fortement le nombre d'octets à traiter.

En contrepartie, l'interprétation des événements bruts est complexe : les méthodes de vision traditionnelles (basées sur des matrices d'intensité) ne s'appliquent pas directement. Il faut convertir les événements en représentations adaptées (images de temps, d'accumulation, cartes de flux optique) et tenir compte du problème d'ego-mouvement : lorsqu'on déplace la caméra, tout le fond génère des événements; il devient difficile de distinguer ce qui est réellement mobile de ce qui correspond au mouvement du capteur.

Algorithme de détection d'objets mobiles

Cette partie résume et structure l'approche proposée dans l'article ICRA 2023 ainsi que les compléments à implémenter. Le pipeline complet comprend quatre étapes : compensation de mouvement, construction d'images temporelles, segmentation par seuillage dynamique et clustering via DBSCAN.

Compensation de mouvement non linéaire

Le cœur de la méthode consiste à « rectifier » les événements afin d'annuler le mouvement de la caméra. On suppose que la rotation dominante est mesurée par une IMU embarquée. Soit (x, y) les coordonnées d'un événement et t l'horodatage relatif au début du paquet d'événements. L'IMU fournit un vecteur de vitesse angulaire $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)$. Les auteurs ont dérivé une relation non linéaire basée sur la géométrie projective : pour chaque événement, on calcule la rotation $\Delta\theta = \boldsymbol{\omega} \cdot t$, puis on applique le changement de repère via les fonctions tangente et arctangente :

$$X_n \quad ¿\,\mathrm{t}\left(\mathrm{a}\left((x - C_x)/f_x\right) - \Delta\theta_x\right),$$
$$Y_n \quad ¿\,\mathrm{t}\left(\mathrm{a}\left((y - C_y)/f_y\right) - \Delta\theta_y\right),$$

où (C_x, C_y) sont les coordonnées du centre optique et (f_x, f_y) la focale en pixels. Les nouveaux points (X_n, Y_n) sont ensuite convertis en indices d'image. Cette formule est plus précise qu'une approximation linéaire, notamment pour des rotations rapides et pour des pixels situés en bordure du capteur. Les tests expérimentaux montrent un gain de précision de 10–15 % par rapport aux approches linéaires 【86159226765082†L129-L158】.

Construction des images temporelles

Après rectification, les événements de fond s'alignent. On construit deux représentations :

- **Time Image** : pour chaque pixel, on calcule la moyenne des horodatages des événements qui y sont projetés. Les pixels du fond ont des temps moyens plus anciens (ils reçoivent beaucoup d'événements tout au long du paquet), tandis que les objets en mouvement ont des temps récents puisqu'ils ne sont observés qu'à un instant donné.
- **Count Image** : on compte le nombre d'événements attribués à chaque pixel. Cette image facilite la visualisation et peut servir de poids pour la segmentation.

Ces images servent d'entrée à l'étape de segmentation.

Segmentation par seuillage dynamique

Pour extraire les objets mobiles, l'article propose un seuil dynamique λ dépendant de la norme de la vitesse angulaire ¿$\boldsymbol{\omega}$‖¿¿ : si le temps moyen T_{ij} d'un pixel (i, j) est supérieur à λ, le pixel est marqué comme appartenant à un objet dynamique. Le seuil est adapté pour tenir compte des variations de vitesse ; de manière générale, plus la rotation est rapide, plus on augmente le seuil afin de ne pas confondre le fond avec les artefacts dus à l'ego-mouvement. On obtient ainsi une carte binaire séparant le fond et les régions dynamiques.

Clustering par DBSCAN

Le nuage de pixels dynamiques résultant de la segmentation contient du bruit. Pour regrouper les points appartenant au même objet, l'article utilise l'algorithme DBSCAN. Ce clustering est basé sur la densité : il identifie les points ayant au moins k voisins dans un rayon ε comme des *points centraux*, relie entre eux les points densément connectés et marque les points isolés comme du bruit【322056068821967†L328-L334】. Le DBSCAN a l'avantage de ne pas imposer un nombre de clusters prédéterminé et de tolérer des formes arbitraires. Dans cette application, les paramètres (ε, k) peuvent être ajustés en fonction de la distance estimée (par la taille apparente des objets) et du bruit du capteur.

Perspectives : suivi et évitement

L'article se concentre sur la détection instantanée. Pour réaliser un système d'évitement d'obstacles, il sera nécessaire d'ajouter des étapes de suivi dans le temps (tracking) afin d'estimer la vitesse et la trajectoire des objets, puis de générer une commande de contournement pour le robot. Des algorithmes comme le filtre de Kalman ou des trackers basés sur la corrélation d'événements pourront être explorés.

Migration de ROS 1 vers ROS 2

Le code fourni dans rpg_dvs_ros était destiné à ROS 1. Pour l'intégrer au robot Limo qui utilise ROS 2 (Humble), un portage complet est nécessaire. Ce chapitre recense les principales différences entre les deux versions et fournit un guide pratique.

Système de build

@ll@ Aspect & ROS 1 (catkin) & ROS 2 (ament_cmake)
Outil de compilation & `catkin_make` ou `catkin_build` & `colcon build`
Fichier principal & `CMakeLists.txt` & `CMakeLists.txt` avec find_package(ament_cmake)
Gestion des dépendances & `find_package(...)` & `find_package(...)` suivie de `ament_target_dependencies`
Installation & `install(TARGETS ...)` à la main dans ROS 1 & Installation automatique via ament pour que `ros2 run` trouve l'exécutable
Export des commandes & optionnel (pas toujours généré) & possibilité de générer `compile_commands.json` pour IntelliSense

Pour porter un paquet, on remplace les instructions catkin par `ament_cmake`. On doit lister explicitement toutes les dépendances dans `find_package` puis les lier avec `ament_target_dependencies`. Enfin, on veille à ce que l'exécutable soit installé dans le dossier `install/lib` via `install(TARGETS ... DESTINATION lib/$PROJECT_NAME)`. Sans cette installation, la commande `ros2 run` ne trouve pas l'exécutable.

Structure des nœuds et paramètres

Dans ROS 1, un nœud est généralement une fonction `int main` qui instancie un `ros::NodeHandle` et crée des publishers, subscribers et services. Les paramètres sont stockés dans un *paramètre server* global. En ROS 2, on favorise l'héritage de la classe `rclcpp::Node`, ce qui permet d'encapsuler logiquement les abonnements et publications. Les paramètres ne sont plus globaux : chaque nœud doit déclarer ses paramètres dans son constructeur à l'aide de `this->declare_parameter(...)` puis les récupérer avec `this->get_parameter(...)`. La méthode `get_parameter` renvoie un objet `rclcpp::Parameter` qui doit être converti explicitement (`.as_int()`, `.as_double()`, etc.)【932557426256562†L286-L303】. Les déclarations de paramètres garantissent que le système peut valider les types et fournir une aide via les fichiers YAML.

Qualité de service (QoS)

ROS 2 introduit la notion de politiques de communication. Un topic n'est connecté que si les QoS du publisher et du subscriber sont compatibles. Pour les capteurs haute fréquence (IMU, événements, images), on utilise généralement le profil `rclcpp::SensorDataQoS` qui définit une fiabilité *BEST_EFFORT* et une profondeur adaptée. Ce profil accepte que certains messages soient perdus afin de ne pas bloquer la pipeline. Les messages de commande ou de contrôle préfèrent une fiabilité *RELIABLE* et une petite file d'attente.

Namespaces et remappages

Les noms de topics en ROS 2 sont résolus en tenant compte du namespace du nœud. Utiliser un nom commençant par `~/` revient à le préfixer par le nom du nœud, ce qui est pratique pour éviter les collisions. Les remappings se font soit dans le code (via `remappings` dans un launch file), soit en ligne de commande. Comprendre les règles de résolution des noms est essentiel pour connecter correctement un driver (par exemple libcaer_driver qui publie sur `~/events` et `~/imu`) avec un nœud de traitement.

Packages ROS 2 pour les caméras événementielles

Le groupe ros-event-camera maintient les paquets compatibles ROS 2 :

- **libcaer_driver** : pilote pour caméras DAVIS et DVXplorer; publie des messages sur `~/events`, `~/imu` et `~/image_raw`.
- **event_camera_msgs** : définitions de messages efficaces pour les événements; ce paquet remplace avantageusement `dvs_msgs` en ROS 2. Toutefois, `dvs_msgs` est toujours disponible pour assurer la compatibilité ascendante.
- **event_camera_renderer** : fournit des outils permettant d’afficher les événements dans RViz ou sur l’écran.

Le tableau 4.1 résume la correspondance entre les paquets ROS 1 et leurs équivalents ROS 2.

Migration des paquets liés aux caméras événementielles

Paquet ROS 1	Paquet ROS 2 équivalent
`davis_ros_driver`	`libcaer_driver`
`dvs_msgs`	`event_camera_msgs` (ou `dvs_msgs`)
`dvs_renderer`	`event_camera_renderer`

Exemple de structure de nœud en ROS 2

Le code suivant illustre la création d’un nœud de compensation en ROS 2. On dérive de `rclcpp::Node`, on déclare les paramètres et on crée les abonnements avec un QoS adapté.

```
class MotionCompensationNode : public rclcpp::Node {
public:
  MotionCompensationNode() : Node("motion_compensation") {
    // Déclaration des paramètres avec valeurs par défaut
    this->declare_parameter<int>("height", 260);
    this->declare_parameter<int>("width", 346);
    this->declare_parameter<double>("focus", 6550.0);
    // Récupération des paramètres
    this->get_parameter("height", height_);
    this->get_parameter("width",  width_);
    this->get_parameter("focus", focus_);
    // Création des abonnements avec QoS SensorData
    auto qos = rclcpp::SensorDataQoS();
    imu_sub_ = this->create_subscription<sensor_msgs::msg::Imu>(
```

```
      "~/imu", qos, std::bind(&MotionCompensationNode::imuCallback, this,
std::placeholders::_1));
    events_sub_ = this->create_subscription<dvs_msgs::msg::EventArray>(
      "~/events", qos, std::bind(&MotionCompensationNode::eventsCallback,
this, std::placeholders::_1));
    image_pub_ =
this->create_publisher<sensor_msgs::msg::Image>("~/image_raw", 10);
  }
private:
  void imuCallback(sensor_msgs::msg::Imu::SharedPtr msg);
  void eventsCallback(dvs_msgs::msg::EventArray::SharedPtr msg);
  // ...
  rclcpp::Subscription<sensor_msgs::msg::Imu>::SharedPtr imu_sub_;
  rclcpp::Subscription<dvs_msgs::msg::EventArray>::SharedPtr events_sub_;
  rclcpp::Publisher<sensor_msgs::msg::Image>::SharedPtr image_pub_;
  int height_{};
  int width_{};
  double focus_{};
};
```

Cette structure clarifie la séparation des responsabilités : le constructeur gère la configuration, les callbacks s'occupent du traitement des messages, et les variables membres remplacent les globales de la version ROS 1.

Programmation C++ moderne

Au cours du stage, de nombreux points de langage ont été abordés. Ce chapitre résume les notions essentielles pour écrire un code fiable et performant en C++.

RAII et gestion des ressources

Le paradigme *Resource Acquisition Is Initialization* (RAII) consiste à gérer l'acquisition et la libération des ressources dans les constructeurs et destructeurs d'objets. Ceci garantit que les ressources (mémoire, fichiers, mutex) sont libérées même en cas d'exception ou de retour anticipé. Par exemple, `std::lock_guard` verrouille un `std::mutex` à la construction et le libère automatiquement à la destruction. En utilisant RAII, on évite les fuites et les blocages que pourrait provoquer un oubli d'appel `unlock()`. Il est recommandé de préférer `std::lock_guard` ou `std::unique_lock` à `mutex.lock()` / `mutex.unlock()`.

Multithreading et synchronisation

Le traitement des événements et des données IMU se fait dans des callbacks exécutés potentiellement en parallèle. Pour éviter les conditions de course, on protège les buffers partagés avec des `std::mutex`. L'utilisation de `std::lock_guard` garantit la libération du verrou. On peut également recourir à `std::unique_lock` lorsque l'on souhaite verrouiller et déverrouiller à des moments précis ou utiliser un verrou conditionnel.

Pour améliorer les performances, il est conseillé de limiter les calculs coûteux (trigonométrie)

dans les boucles et de regrouper les copies de données. L'utilisation d'un `rclcpp::executors::MultiThreadedExecutor` permet à ROS 2 de lancer plusieurs callbacks en parallèle en fonction du nombre de cœurs disponibles.

std::bind et lambdas

Lors de la création de subscribers, on transmet une fonction de rappel. Deux approches existent :

- `std::bind` : lie les arguments et l'instance de la classe. Par exemple : `std::bind(&Class::method, this, _1)`.
- **Lambdas capturant** `this` : plus concis `[this](auto msg) { this->method(msg); }`.

Les lambdas sont recommandées pour leur lisibilité et car elles permettent d'éviter les erreurs de type. L'essentiel est d'associer correctement la méthode à l'instance actuelle.

Gestion de la mémoire et performances

La compensation et le clustering nécessitent de parcourir des dizaines de milliers d'événements par seconde. Pour maintenir le temps réel, on veille à :

- Éviter les allocations dynamiques répétées à l'intérieur des boucles (réserver la mémoire des vecteurs).
- Utiliser des types natifs (`float`, `double`) et éviter les conversions inutiles.
- Exploiter l'optimisation du compilateur (options `-O3`) et, si nécessaire, paralléliser les boucles avec OpenMP ou TBB.

Environnement de développement et outils

Configuration de VS Code

L'IDE VS Code est pratique pour développer en ROS 2 grâce à ses extensions. Toutefois, l'autocomplétion (IntelliSense) doit savoir où chercher les fichiers d'en-tête. Après compilation avec `colcon build`, les headers des paquets installés se trouvent dans `install/<package>/include`. On ajoute ce chemin dans `.vscode/c_cpp_properties.json` :

```
{
    "configurations": [
        {
            "name": "Linux",
            "includePath": [
                "${workspaceFolder}/**",
                "${workspaceFolder}/install/**/include",
                "/opt/ros/humble/include/**"
            ],
            "compilerPath": "/usr/bin/gcc",
```

```
            "cStandard": "c17",
            "cppStandard": "gnu++20",
            "intelliSenseMode": "linux-gcc-x64"
        }
    ],
    "version": 4
}
```

On peut également générer un fichier `compile_commands.json` en passant l'option `–cmake-args -DCMAKE_EXPORT_COMPILE_COMMANDS=ON` à `colcon build`. Ce fichier permet à l'IDE de connaître exactement les arguments de compilation de chaque fichier source.

Utilisation de libcaer_driver

Pour utiliser la DAVIS 346 en ROS 2, on installe le paquet `libcaer_driver` via apt ou à partir des sources. Le driver publie plusieurs topics privés (`~/events`, `~/imu`, `~/image_raw`). Il est lancé via un fichier launch qui permet de configurer le type de caméra, l'identifiant, l'activation de l'APS et de l'IMU, et de remapper les topics si besoin. Voici un extrait de lancement :

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, OpaqueFunction
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def launch_setup(context, *args, **kwargs):
    node = Node(
        package='libcaer_driver',
        executable='driver_node',
        name=LaunchConfiguration('camera_name'),
        parameters=[{
            'device_type': LaunchConfiguration('device_type'),
            'dvs_enabled': True,
            'imu_accel_enabled': True,
            'imu_gyro_enabled': True,
            'aps_enabled': False,
            'event_message_time_threshold': 1.0e-3,
        }],
        remappings=[('~/reset_timestamps',
LaunchConfiguration('reset_topic'))],
    )
    return [node]

def generate_launch_description():
    return LaunchDescription([
        DeclareLaunchArgument('camera_name', default_value='event_camera'),
        DeclareLaunchArgument('device_type', default_value='davis'),
        DeclareLaunchArgument('reset_topic',
default_value='~/reset_timestamps'),
        OpaqueFunction(function=launch_setup),
    ])
```

Ce fichier montre que le nom réel du nœud est déterminé par `camera_name`; les topics préfixés par `~/` seront donc résolus en `/event_camera/events`, etc. Comprendre ce comportement est

indispensable pour connecter correctement vos abonnements.

Plan de travail et recommandations

L'organisation proposée lors des échanges prévoit un déroulement en plusieurs phases :

1. **Semaine 1–2** : portage du code ROS 1 vers ROS 2, création d'un paquet `datasync` avec la nouvelle structure, déclaration des paramètres, mise en place du QoS `SensorDataQoS`, compilation avec colcon, vérification des exécutions avec un bag de test.
2. **Semaine 3–4** : intégration sur le robot Limo : installer libcaer_driver sur la machine embarquée, calibrer l'alignement IMU/caméra, optimiser le code pour le processeur embarqué (réduction des appels trigonométriques, utilisation de pointeurs bruts, etc.).
3. **Semaine 5–6** : implémentation de la segmentation dynamique, réglage du seuil en fonction de la vitesse, premier test de clustering avec DBSCAN; évaluation de la précision et du temps d'exécution.
4. **Semaine 7–8** : ajout d'un suivi temporel des objets, amélioration de la robustesse (filtrage du bruit, fusion de clusters), développement d'une stratégie de navigation pour contourner les obstacles détectés.

Cette planification est indicative et peut être adaptée en fonction des contraintes de terrain (mise à disposition du robot, fiabilité des drivers, etc.).

Conclusion et perspectives

Le travail de capitalisation effectué à travers ce rapport offre un panorama complet des connaissances nécessaires pour développer une chaîne de détection d'obstacles dynamiques avec une caméra événementielle sous ROS 2. Nous avons exploré les principes des capteurs bio-inspirés, détaillé les étapes de compensation de mouvement non linéaire et de regroupement des événements, étudié la migration de ROS 1 vers ROS 2 ainsi que les outils de développement associés. Des notions transverses de programmation C++ moderne (RAII, multithreading, lambdas) ont été mises en avant afin de produire un code robuste et maintenable.

La suite du projet consistera à implémenter les briques manquantes (segmentation, clustering), à optimiser l'algorithme pour des plateformes embarquées et à intégrer un module de navigation. Grâce à cette base solide, le stagiaire dispose d'un véritable manuel de référence pour affronter les problèmes rencontrés et réutiliser les solutions décrites.