

Option Réalité virtuelle - TP Réalité Augmentée

Module VSION

Jean-Marie NORMAND

15 décembre 2025 et 05 janvier 2026

L'objectif de ce TP est de construire une petite application de réalité augmentée en C++. Pour cela, nous nous appuierons sur deux bibliothèques externes :

- OpenCV, que vous avez déjà utilisé ;
- ArUco, qui est une petite bibliothèque de réalité augmentée s'appuyant sur OpenCV.

Vous pouvez (devez?) revoir les transparents du cours sur la réalité augmentée de FONRV. Vous devrez nous rendre un compte-rendu à la fin de la séance contenant **obligatoirement une adresse vers un dépôt Git** où se trouveront les sources de votre code.

1 ARUCO : PREMIER PROGRAMME

Dans cette partie, nous allons construire une première application. Celle-ci se contentera de détecter des marqueurs. À partir des points caractéristiques des marqueurs, ArUco sera par la suite capable de calculer la position et l'orientation de la caméra.

Pour construire cette première application, vous devez :

1. Télécharger et compiler la bibliothèque ArUco 3.1.12 sur Hippocampus
2. Créer et **configurer** un projet "Application Console Win 32" comportant un simple fichier source ainsi qu'ArUco et OpenCV.
3. Écrire un morceau de code qui va lire une image dont le nom sera passé en ligne de commande et appeler les méthodes de détection de marqueurs d'ArUco à l'aide des extraits de code ci-dessous.
4. Créer un marqueur avec un exécutable créé lors de la compilation d'ArUco :*aruco_create_marker*. Le marqueur peut être imprimé ou filmé directement sur l'écran de votre poste de travail.

```

// creation d'un detecteur de marqueurs
MarkerDetector myDetector;

// liste de marqueurs : sera remplie par ArUco

//detect markers and for each one, draw info and its boundaries in the image
for (auto m : myDetector.detect(myImage)) {
    std::cout << m << std::endl;
    m.draw(myImage);
}

```

Dans le code ci-dessus, *myImage* est une image de type *cv : Mat* contenant l'image d'intérêt (par exemple le flux de la webcam de votre PC, voir TPs précédents).

La boucle sert à afficher l'ensemble des marqueurs détectés, comme illustré dans la Figure 1.1 (qui n'en contient qu'un).

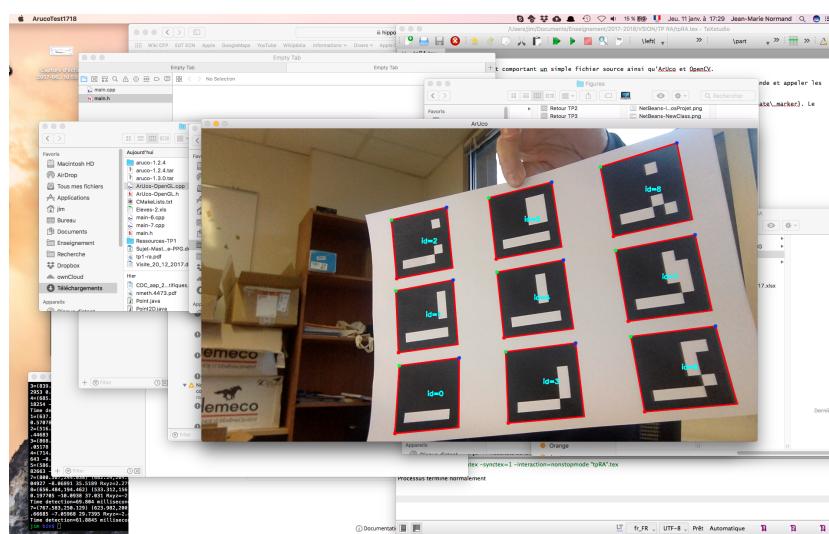


FIGURE 1.1 – Exemple de détection de 9 marqueurs ArUco.

Étudiez et reportez le comportement d'ArUco vis-à-vis du ou des marqueurs, que se passe t'il lorsque :

- a un ou plusieurs marqueurs,
- quand un marqueur est plus ou moins visible,
- l'angle de vue de la caméra est changé,
- la taille du ou des marquer(s) change sur l'image,
- etc.

2 COMPILER ARUCO 3.1.12 AVEC CMAKE

Nous allons profiter de ce TP pour vous faire utiliser CMake pour compiler une bibliothèque sous Windows. Pour ce faire, nous allons :

1. décompresser l'archive fournie ([ArUco-3.1.12](#)) dans le répertoire de téléchargements;
2. aller dans le répertoire racine (celui qui contient les répertoires `3rdparty`, `src`, `utils` etc.) et **créer un nouveau dossier** que vous appellerez `build`;
3. lancer CMake et dans la fenêtre CMake vous mettrez (voir Figure 2.1) :
 - comme chemin vers le code source : [chemin-vers-ArUco3.1.12](#)
 - comme chemin *where to build the binaries* : [chemin-vers-ArUco3.1.12/build/x64](#) (le dernier répertoire - x64 - sera créé automatiquement)
4. cliquer sur [Configure](#) (choisir “Visual Studio 16 2019” et 64 bits **x64** dans la fenêtre de choix des configurations);
5. des erreurs s'affichent : **c'est normal** :) Nous allons les corriger et configurer correctement CMake.
6. configurer CMake comme le montrent les Figures 2.2 et 2.4 (attention aux lignes en surbrillance!). Notez en particulier qu'il faut que vous fournissiez comme “**OpenCV_DIR**” un dossier où OpenCV est installé et dans lequel un fichier “OpenCV-Config.cmake” est présent!;
7. cliquer sur [Configure](#) : une erreur est présente :D Il faut aller lui dire où trouver GLUT, configurer le comme illustré en Figure ??;
8. cliquer (encore) sur [Configure](#) : tout est bon cette fois-ci o/;
9. cliquer sur [Generate](#) : il peut y avoir du rouge mais normalement tout s'est bien passé;
10. aller dans le répertoire [chemin-vers-ArUco3.1.12/build/x64](#), et double-cliquer sur le fichier `aruco.sln` (attention à bien choisir le bon fichier!);
11. attendre l'ouverture de Visual Studio puis double cliquer sur le **projet** : **ALL_BUILD** (normalement il est déjà sélectionné en gras), puis cliquer droit dessus et faire : [Générer](#);
12. croiser les doigts que tout se passe bien et si c'est le cas vous verrez à la fin de la compilation “Génération : 21 a réussi” si c'est le cas BRAVO!
13. cliquer droit sur le projet : [INSTALL](#) et faites [Générer](#).

Dans votre rapport :

- ajoutez un paragraphe qui résume ce que vous avez compris des étapes précédentes et de l'intérêt d'un outil comme CMake;
- expliquez où vont s'installer les éléments compilés et pourquoi là bas?
- détaillez quels sont les exécutables et quelles sont les bibliothèques qui ont été créées dans le processus;
- expliquez comment vous allez maintenant utiliser Aruco dans votre projet.

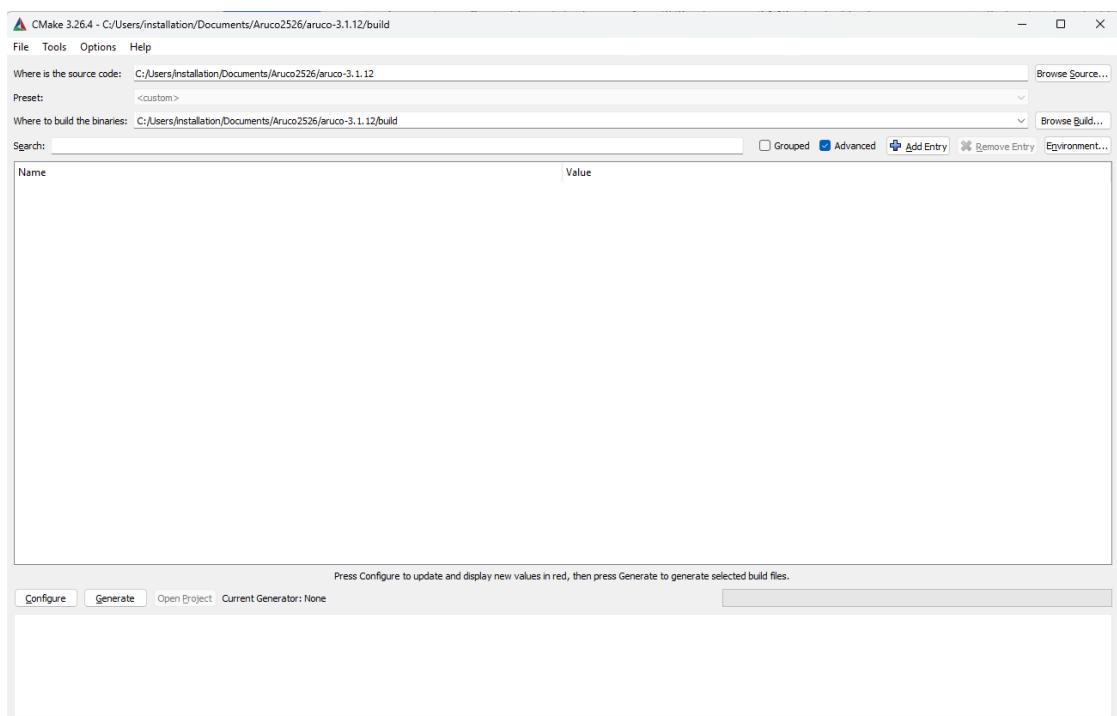


FIGURE 2.1 – Configuration CMake - 1re partie.

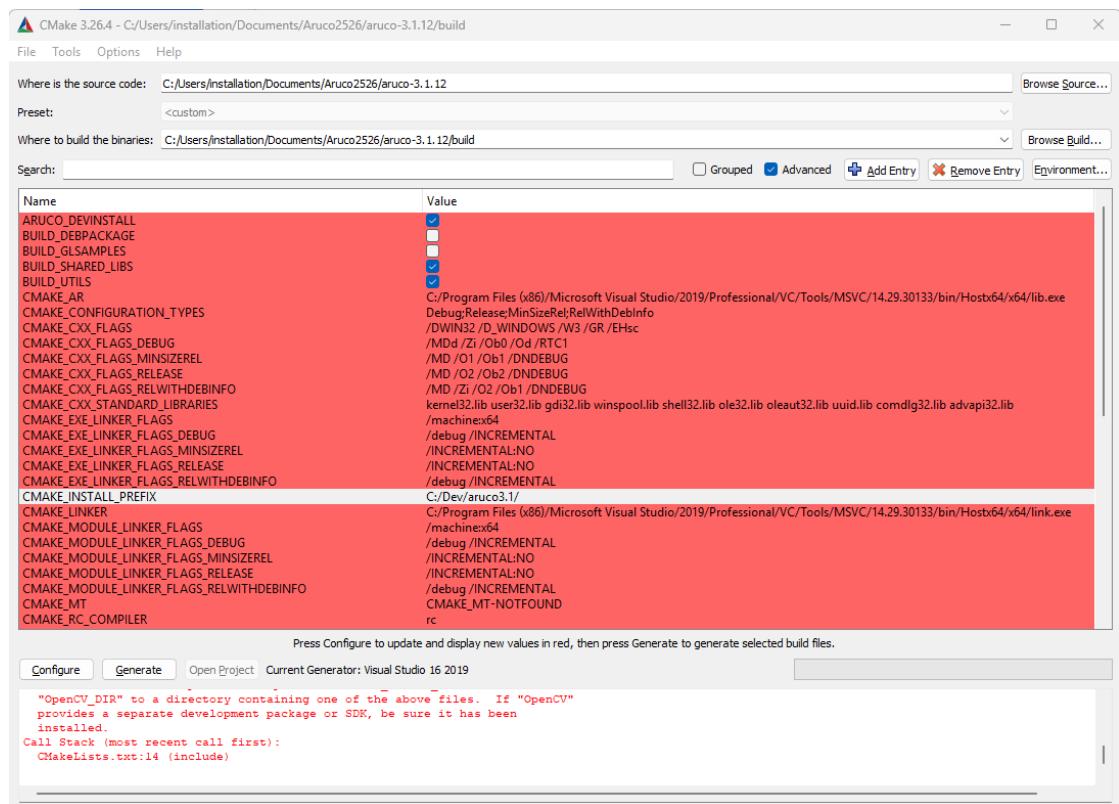


FIGURE 2.2 – Configuration CMake - 2e partie.

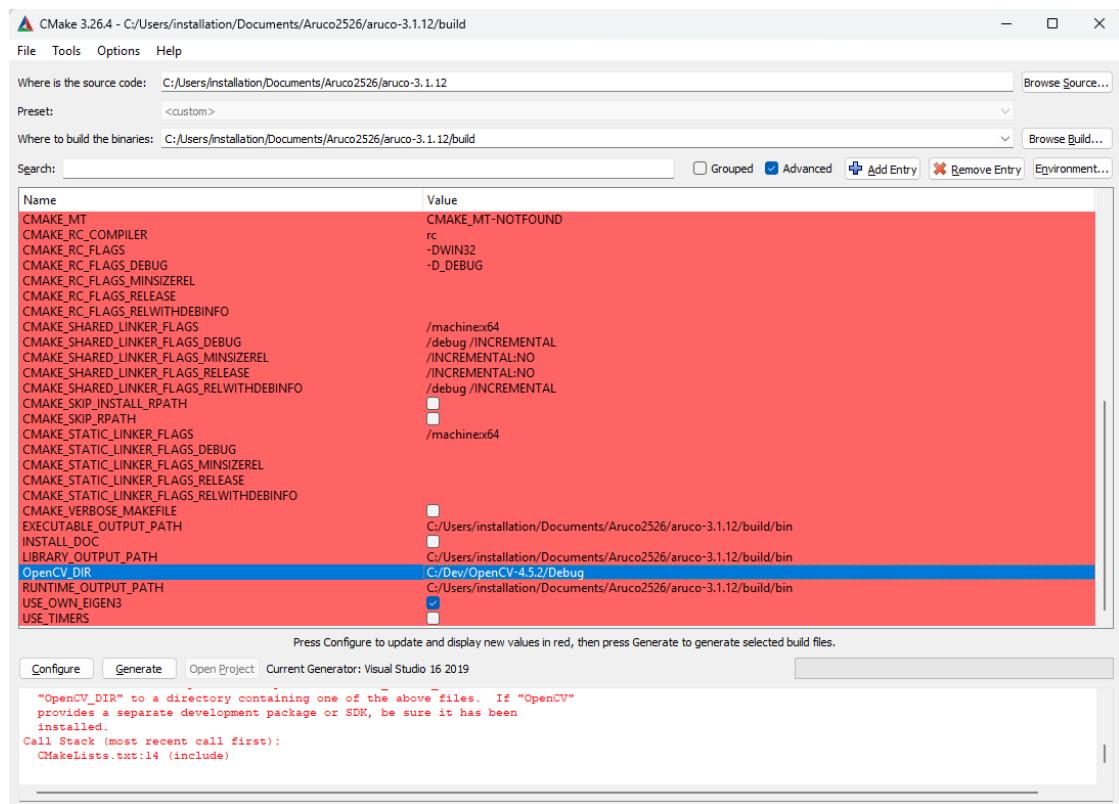


FIGURE 2.3 – Configuration CMake - 3e partie.

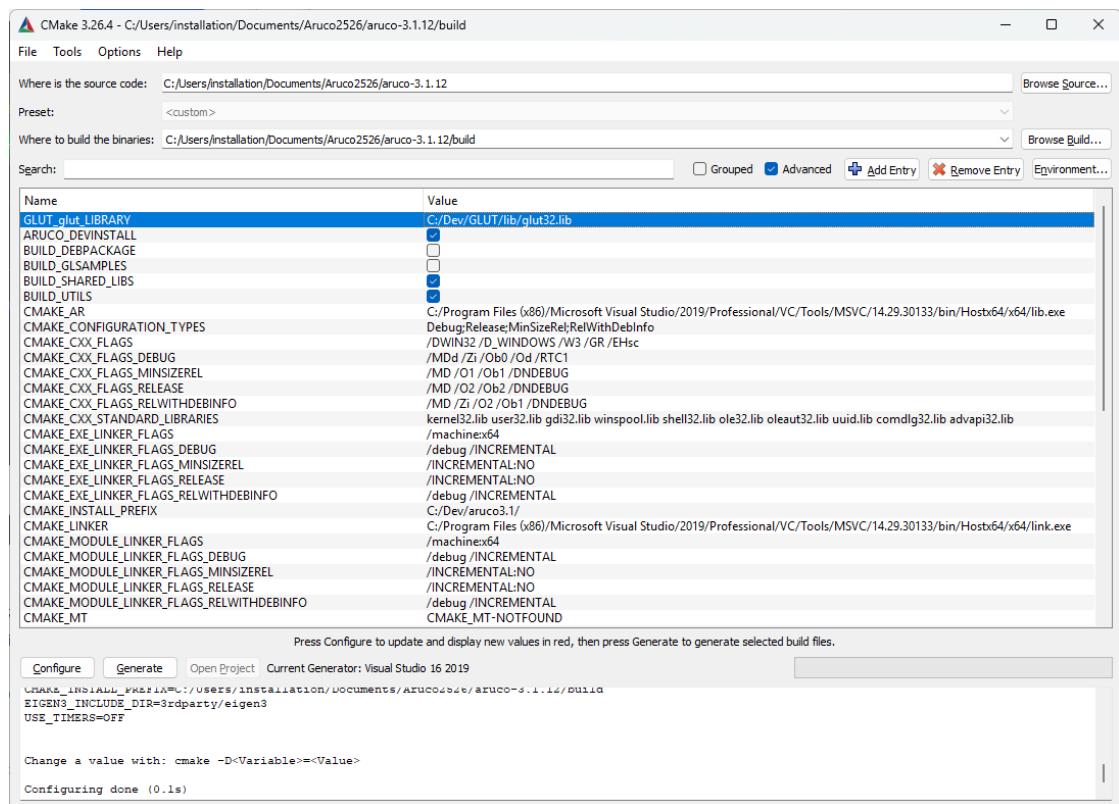


FIGURE 2.4 – Configuration CMake - 4e partie.

3 PREMIÈRE AUGMENTATION

Pour commencer, nous allons utiliser une petite application basée sur OpenGL/GLFW pour détecter les marqueurs et générer des augmentations simples.

Le code de celle-ci est disponible sur le serveur pédagogique et se compose de plusieurs fichiers :

- *main.cpp* qui sert essentiellement à effectuer les initialisations nécessaires, notamment pour GLFW. Il définit également une instance de la classe ArUco définie dans les fichiers *ArUco-OpenGL.h* ainsi que quelques fonctions de callback. Certaines, comme la gestion du clavier ou de la souris sont largement vides à ce stade.
- *ArUco-OpenGL.h* et *ArUco-OpenGL.cpp* : ce second groupe de fichiers est consacré à la définition de la classe *ArUco*. Les fonctions de cette classe servent à mettre en œuvre les fonctions d'affichage graphique à l'aide d'OpenGL et des fonctions de détections de marqueurs d'Aruco.

La méthode la plus importante pour vous est la méthode *ArUco* : *:drawScene()* : c'est elle qui est chargée de dessiner à la fois le fond vidéo et les augmentations.

Dans la version qui vous est fournie, elle initialise les différents éléments nécessaires avec OpenGL. J'ai ajouté de nombreux commentaires en espérant vous faire comprendre ce que fait le code. De plus, j'ai dessiné sur chaque marqueur un cube en file de fer pour illustrer l'utilisation d'un dessin OpenGL.

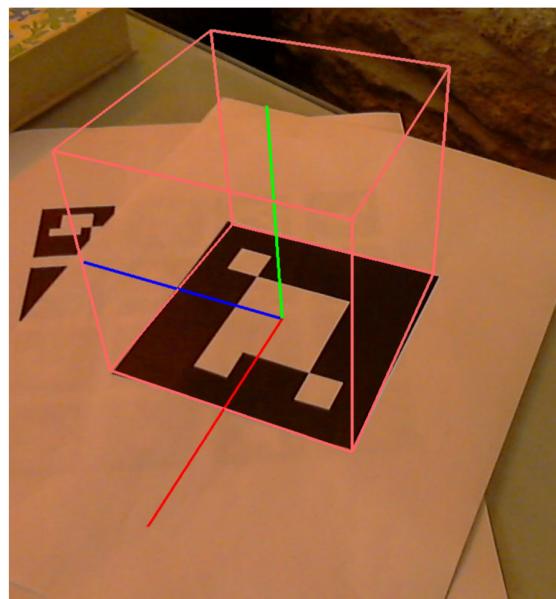


FIGURE 3.1 – Première augmentation en ArUco-OpenGL.

Le travail à effectuer pour cette partie est le suivant :

1. Récupérer et compiler cette première application.

2. Détaillez là où vous avez mis les fichiers liés à GLFW (fournis sur Hippocampus). Expliquer pourquoi vous les avez mis à cet endroit sur le disque dur.
3. Si vous avez des problèmes de link (édition des liens) ou d'exécution (dll manquantes?) : explicitez les erreurs rencontrées et expliquez comment vous les avez résolues.
4. Vérifier qu'elle fonctionne. Vous devez fournir au programme le fichier *camera.yml* réalisé lors du TP calibration (ou celui que je fournis) ; incluez une copie d'écran dans votre rapport.
5. Expliquer ce que vous comprenez des enchaînements d'actions OpenGL de la méthode *ArUco ::drawScene()*.
6. Supprimer les augmentations actuelles des marqueurs qui servent à vérifier que tout fonctionne et dessinez un objet différent de votre choix à l'aide d'OpenGL. Vous pouvez récupérer du code que vous avez écrit pendant les cours d'IMAGRIV si vous le souhaitez. Vérifier que l'augmentation est stable quand on bouge la caméra ou le marqueur.
7. Joindre au rapport le code de la méthode *ArUco ::drawScene()* ainsi que les éventuelles nouvelles fonctions.

4 APPLICATION DE RÉALITÉ AUGMENTÉE – RA

Dans cette partie, vous allez devoir imaginer une petite application de réalité augmentée. Celle-ci devra utiliser au moins deux marqueurs et proposer une certaine interaction liée au mouvement des marqueurs. Vous pouvez travailler l'éclairage, l'interaction, construire un petit jeu, etc.

Le rapport devra comporter :

- Le code de l'application et tous les fichiers nécessaires à son bon fonctionnement.
- L'objectif de l'application et son degré de réalisation (i.e. ce qui marche vs. ce qui ne marche pas encore).
- Le mode d'emploi de l'application.

5 IDÉE D'EXEMPLE : LE RENDU FANTÔME EN OPENGL

L'idée est d'aller plus loin sur l'intégration entre le monde virtuel et le monde réel en prenant en compte les occultations qui devraient exister entre les deux objets des deux mondes. Techniquement, les choses sont extrêmement simples à mettre en œuvre :

```
// Desactivation du color buffer pour le modèle du monde réel
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

// affichage du modèle virtuel du monde réel uniquement dans le depth bufer
displayVirtualModelOfRealWorld();

// Reactivation du color buffer pour dessiner les objets virtuels (e.g. voiture)
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
```

```
// on dessine maintenant les objets virtuels  
displayVirtualWorld();
```

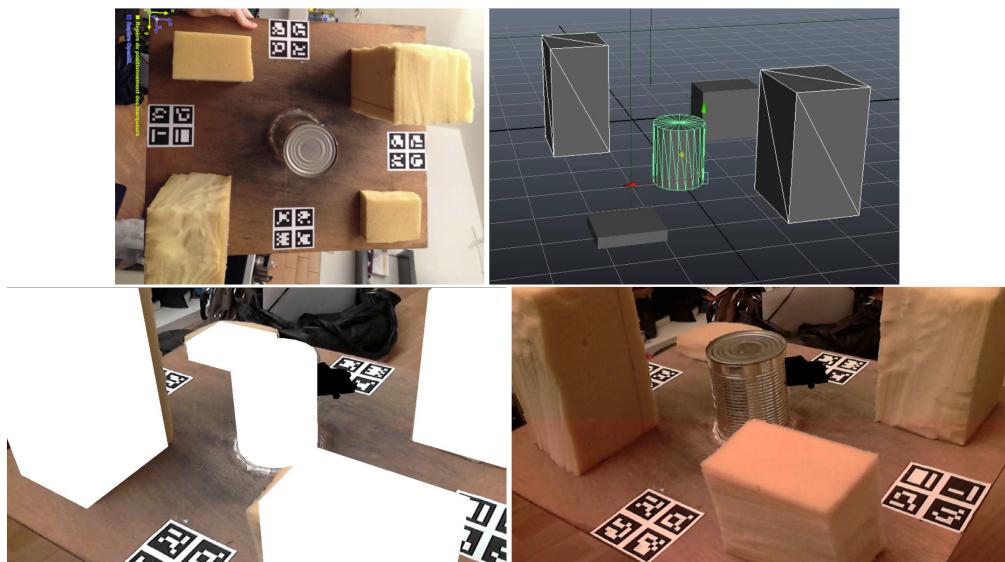


FIGURE 5.1 – Monde réel, modèle virtuel du monde réel, rendu fantôme du modèle du monde réel, rendu final avec occultations correctes de la voiture virtuelle par les objets réels.