Master 2 TSI - Option Programmation

TP 1 de Réalité Augmentée

Au cours de ce TP, nous développerons une application simple de réalité augmentée pour afficher des éléments 3D à l'aide de tags imprimés.

Instructions préliminaires : téléchargez le dossier de code en utilisant le dépôt git :

- `git clone https://github.com/npiasco/ENSG_AR_TP1.git`

Compilez séparément la librairie apriltag (lire le `README` dans `ENSG_AR_TP1/external/apriltag`).

Exécuter le scripte `dependency.sh` pour installer les bibliothèques nécessaires à la compilation du projet.

Générez le **Makefile** du projet :

- `cd ENSG_AR_TP1`
- `mkdir build && cd build`
- `cmake ..`

Compilez le projet :

- `make -j4`

1 Calibration de la caméra

Problématique : comme nous l'avons vu dans le cours d'introduction précédent, nous allons avoir besoin d'une caméra calibrée pour réaliser notre application de réalité augmentée. Les webcams que nous utilisons présente une *faible* distorsion, nous allons donc considérer que la phase de calibration servira seulement à déterminer les paramètres intrinsèques de focal et de position du centre optique de notre caméra.

1. Branchez la webcam à votre ordinateur et lancez l'exécutable `calibration` dans le dossier `ENSG_AR_TP1/opencv_code/calib/` pour procéder à la calibration. Quelles sont les paramètres intrinsèques de votre caméra ?

- fx =
- fy =
- cx =
- cy =

Note : si l'autofocus de la webcam est fonctionel, vous pouvez le désactiver avec la commande `v4l2-ctl -c focus_auto=0` (package `v4l-utils`).

Le champ de vue d'une caméra (FoV) est défini par :

$$FoV_{digonal} = 2 * arctan\left(\frac{d}{2 * f}\right)$$

Avec :

— d = la taille de la diagonal du capteur
— f = la distance focale

2. En prenant en compte les paramètres de calibration obtenus, calculez le champ de vue de votre caméra.

— FoV horizontal =
— FoV vertical =

3. Après avoir vu le retour image de la caméra, pourquoi peut-on dire que la distorsion induite par la lentille peut être négligée ?

2 Prise en main des AprilTags

Les **Apriltags** vont nous servir de balises dans notre repère monde pour positionner de façon précise les éléments 3D que nous allons rajouter à l'environnement.

1. Lancez le programme `mainCV` situé dans le dossier `ENSG_AR_TP1/opencv_code/` pour vérifier que les tags sont bien détectés.

On a mis à votre disposition une classe `AprilTagReader` dans le dossier `ENSG_AR_TP1/opencv_code/`.

2. Lisez la documentation présente dans le fichier `AprilTagReader.h`. Écrivez un programme dans le fichier `AprilTagReaderTest.cpp` permettant d'afficher la position des tags détectés par la classe.
3. En vous basant sur vos observations, dessinez le repère associé à la caméra dans lequel est représenté la position des tags selon la classe `AprilTagReader` :

3 Simulation de la caméra dans OpenGL

Nous allons écrire le code principale de notre programme dans le fichier `main.cpp` présent dans le dossier `ENSG_AR_TP1`. Il s'agit d'un fichier reprenant la structure d'un code **OpenGL** : en effet *la boucle principale* de notre application de réalité augmentée se situera dans la boucle d'affichage habituelle d'OpenGL.

1. Familiarisez vous avec la structure du code OpenGL. Les fichiers `Model.h` et `Shader.h`/`Shader.cpp` permettent respectivement de charger des modèles 3D et les shaders du programme.

Nous allons maintenant simuler une caméra dite *projective* (à la différence d'orthogonale) qui aura les même paramètres que notre webcam, ainsi le rendu de nos objets 3D dans la scène capturée par la caméra sera réaliste.

Pour afficher le flux vidéo dans OpenGL, nous allons coller l'image (que l'on récupèrera grâce à la classe `AprilTagReader`) en tant que texture sur un rectangle composé de 4 vertices. Cette méthode est loin d'être optimal mais pour des images de dimension pas trop grande (on limitera la résolution de l'image à $640 * 480$) cela reste faisable. Lancez le programme `main` pour voir le résultat.

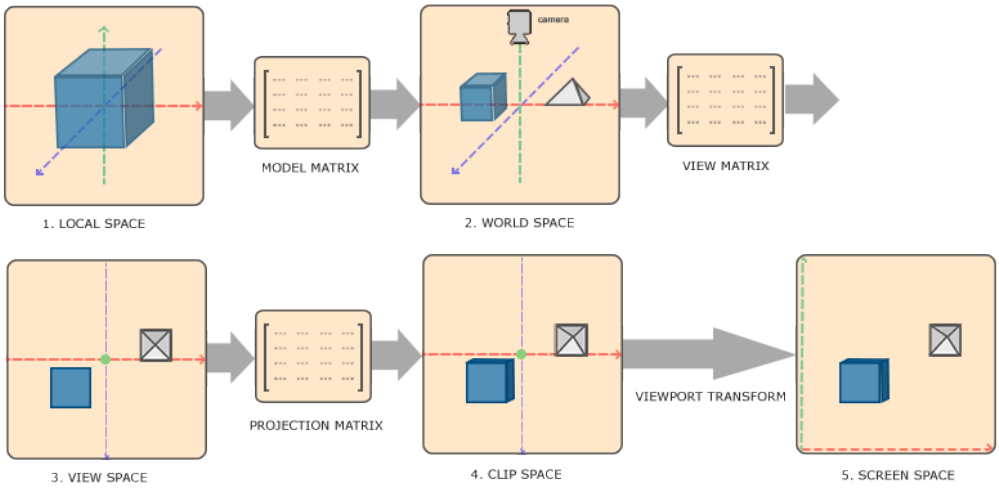

FIGURE 1 – Figure 1 : Rappel des transformations successives avant l'affichage à l'écran de notre scène OpenGL

La figure ci-dessus rappelle les transformations successives opérées sur nos objets 3D pour obtenir le rendu dans notre fenêtre OpenGL. Dans le cas particulier de notre application de réalité augmentée nous pouvons fixer à l'initialisation : la position de la caméra (matrice `view`), la position du rectangle contenant la texture (matrice `texture_model`) et la matrice de projection (`projection`).

2. A l'aide de la fonction `glm::mat4 glm::perspective(GLfloat HFOV, GLfloat aspect_ratio, GLfloat near_plane , GLfloat fare_plan)`, créez une matrice de projection simulant les paramètres de votre webcam. Attention à l'unité de `HFOV`.

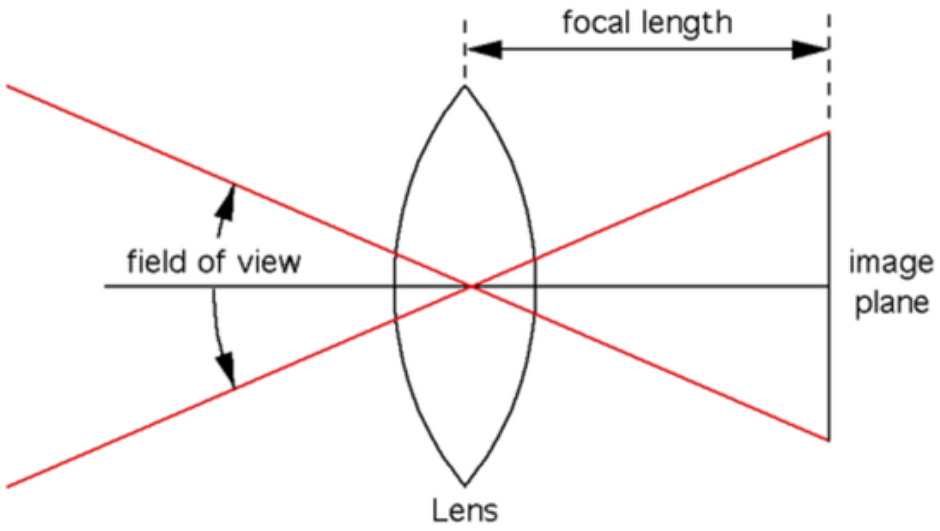

FIGURE 2 – Figure 2 : Schéma d'une caméra pin-hole

On va maintenant modifier les différents éléments de l'environement 3D de tel sorte à aligner le repère de la caméra (partie 2, question 3) avec la position de la caméra fictive d'OpenGL et à remplire l'intégralité du champ de vue avec notre texture de flux vidéo.

3. En vous aidant de la figure 2 **et d'un schéma** modifier les différents élements du code pour parvenir à l'aignement désiré. On pourra notament jouer sur la position de la caméra (fonction `lookAt`), la position de la texture (variable `vertices`) ou appliquer des transformations au model (variable `texture_model`).

4 Affichage des objets dans l'environnement augmenté

Maintenant que notre flux vidéo a été incorporé dans OpenGL, nous pouvons facilement intégrer des modèles 3D pour augmenter notre environnement. L'utilisation d'Apriltag va nous permettre de positionner nos éléments dans notre scène OpenGL.

1. En récupérant la position d'un tag, remplire la matrice `model` traduisant la position de ce tag dans le repère d'OpenGL.

Attention : les matrices `glm` d'OpenGL ont une convention de stockage inverse de celle utilisée par OpenCV pour ranger leurs éléments : `mat[i][j]` fait référence à l'élément de la i^{eme} colonne, j^{eme} ligne.

La classe `Model` permet de charger et d'afficher facilement des modèles 3D (.obj, .3ds, .stl, etc.). Pour charger un modèle indiquez simplement le chemin vers le modèle à charger comme argument du constructeur. Pour afficher le modèle utilisez la méthode de classe `void Draw(Shader shader)`.

2. Affichez le singe de blender Suzanne (`ENSG_AR_TP1/opengl_code/model/suzanne/suzanne.obj`) sur l'apriltag.

Attention : vous devez rafraîchir le buffer de profondeur avec la commande : `glClear(GL_DEPTH_BUFFER_BIT )` au bon moment pour être sur de voir votre modèle (vu qu'il sera sûrement positionné *derrière* le rectangle contenant comme texture l'image de la caméra).

3. Appliquez des transformations de mise à l'échelle et de rotation sur votre modèle 3D afin d'obtenir un affichage de réalité augmentée *convaincant*.

Note : pour appliquer une mise à l'échelle de votre model utilisez la fonction suivante : `glm::mat4 glm ::scale(glm::mat4 m_init, glm::vec3 vect_scale)`. Pour appliquer une rotation à votre model utilisez la fonction suivante : `glm::mat4 glm::rotate(glm::mat4 m_init, GLfloat angle_rad, glm::vec3 vect_rot)`.

5 Aller plus loin

Votre application de réalité augmentée est maintenant fonctionnelle ! Nous vous proposons dans cette partie de l'améliorer en y rajoutant différents modules.

5.1 Affichage de différents objets

Les apriltags nous permettent d'une part de connaître la pose dans l'espace d'une feuille de papier à partir d'une caméra monoculaire et d'autre part de lire le numéro associé à la cible détectée. Modifiez votre code pour permettre l'affichage simultané de différents modèles en fonction du numéro des tags.

5.2 Interface augmentée

L'une des utilisations de la réalité augmentée est de créer des interfaces plus ergonomique pour l'utilisateur. Afin de créer une interface utilisateur, nous allons utiliser trois tags différents : les deux premiers serviront de boutons tandis que le dernier affichera le modèle 3D. En fonction de la distance à la caméra aux deux premiers tags, le modèle 3D sur le troisième tag devra changer de taille et d'orientation.

5.3 Persistance des modèles

Actuellement, la moindre occultation du tag provoque la disparition du modèle associé. Intégrez dans votre code un moyen de maintenir à la dernière position connue l'affichage du modèle même si le tag n'est plus visible un court instant.