L'application

Partie 1.1

Généralitées

1.1.1 Obtenir le code source

Le code source de l'application est disponible en ligne sur GitHub [1]. Ce code disponible est sous licence CeCILL-B [2].

1.1.2 Installation

Développé en C++11, la compilation de l'application nécessite `gcc 4.7` ou supérieure. L'outil `cmake` est par ailleurs utilisé afin d'automatiser les différentes étapes de la compilation et de l'installation.

Pour plus d'informations sur la procédure d'installation, un fichier `README.txt` est disponible avec le code.

1.1.3 Dépendances

Le code développée fait appel à différentes bibliothèques pour remplir des taches spécifiques. À ce titre, l'utilisation de l'application peut demander l'installation préalable des outils utilisés.

OpenCV : Regroupant de nombreuses opérations de traitement d'image, OpenCV [3] est utilisé pour la manipulation des images ainsi que pour les différentes étapes de calibration et les calculs matriciels associés.

Eigen : Utilisée pour le pré calcule des sphères (voir section **??**, page **??**), Eigen [4] est une bibliothèques de calcul matriciel ne nécessitant pas d'installation préalable.

Le framework gKit, développé au LIRIS [5], utilisé dans cette application induit également un certain nombre de dépendances.

GLEW : GLEW [6] est un bibliothèques multiplateforme utilisée pour la gestion des extentions OpenGL.

SDL2 : SDL [7] est une bibliothèque multiplateforme permettant un accès bas niveau aux différentes interfaces utilisateurs ainsi qu'au matériel graphique via OpenGL et Direct3D.

SDL2 Image : SDL2_image [8] est une extension de SDL permettant la gestion d'images dans différents formats.

SDL2 TTF : SDL2_ttf [9] est une extension de SDL permettant la gestion des polices de caractères.

OpenEXR : OpenEXR [10] est un dépendance optionnel utilisé pour la gestion des formats d'image HDR.

Enfin la construction des différents modèles induit aussi certaines dépendances supplémentaires.

OpenCV : En plus des outils de traitement d'image, OpenCV [11] est utilisé par le module du même nom pour la gestion des interfaces d'acquisition vidéo (webcam).

ZBar : La bibliothèque ZBar [12] est utilisé par le module du même nom pour la détection et le décodage des QRCodes présents dans les images issues des cameras.

Partie 1.2

Structure

Écrite en C++11, l'application développée profite de la programmation orientée objet pour organiser le code suivant les différentes fonctionnalités. Afin de rendre l'application modulable, certains composants sont par ailleurs encapsulé suivant des modèles de classes virtuelles et chargé dynamiquement.

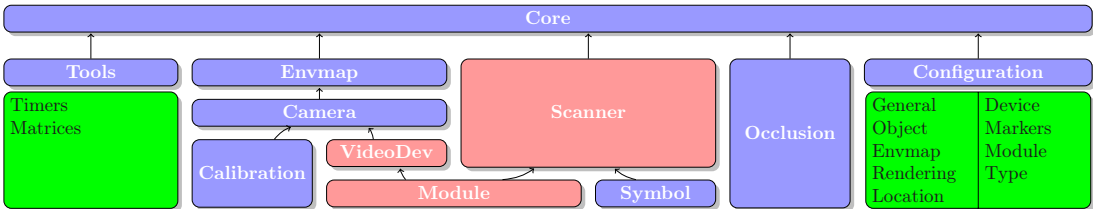
1.2.1 Hiérarchie

Les différentes composants logiciels sont encapsulé dans des objets spécifiques selon le paradigme de programmation orienté objet. La figure 1.1 (page 2) présente cette hiérarchie.

Core : La classe `Core` décrit un instance de l'application. Héritant du modèle `gk::App` présent dans `gKit`, elle

1. https://github.com/Amxx/MobileReality/tree/master/code
2. **CeCILL :** http://www.cecill.info/
3. **OpenCV :** http://opencv.org/
4. **Eigen :** http://eigen.tuxfamily.org/
5. **LIRIS :** http://liris.cnrs.fr/
6. **GLEW :** http://glew.sourceforge.net/
7. **SDL2 :** http://www.libsdl.org/
8. **SDL2 Image :** http://www.libsdl.org/projects/SDL_image/
9. **SDL2 TTF :** http://www.libsdl.org/projects/SDL_ttf/
10. **OpenEXR :** http://www.openexr.com/
11. **OpenCV :** http://opencv.org/
12. **ZBar :** http://zbar.sourceforge.net/

Figure 1.1 Structure de l'application



est crée au lancement du programme et gère l'application graphique ainsi que toute la hiérarchie de classes gérant les différents composants et fonctionnalités.

Module : La classe `Module` est une classe mettant en places les différentes méthodes nécessaire au chargement dynamique d'objet pré-compilés correspondant à différents types de structures. Cette classe permet ainsi une grande flexibilité dans le chargement de composants externes.

VideoDevice : La classe `VideoDevice` est une classe virtuelle décrivant l'interface acquisition vidéo. Cette classe est implémentée par les modules `videodevice_opencv` et `videodevice_uvc` qui constituent l'interface entre l'application et les périphériques d'acquisition vidéo.

Calibration : La classe `Calibration` gèrent les différentes valeur caractérisant un objet de type `VideoDevice`. Elle intègre aussi les méthodes de calcul et d'enregistrement / chargement de ces données de calibration.

Camera : La classe `Camera` regroupe un objet de type `VideoDevice` ainsi qu'un objet de type `Calibration`. Héritant de la structure virtuelle `VideoDevice`, elle permet de gérer un périphérique vidéo calibré de manière transparente.

Scanner : La classe `Scanner` est une classe virtuelle décrivant l'interface relatives aux modules de détection de symboles dans les images. Cette classe virtuelle est implémentée dans le module `scanner_zbar` qui utilise par bibliothèque ZBar [13] pour l'identification de QR-Codes.

Symbol : La classe `Symbol` décrit les symboles identifiées par les modules de type `Scanner`. Contenant les informations de positionnement ainsi que les données encodées, ils permettent de calculer les données de positionnement (voir section **??**, page **??**).

Envmap : La classe `Envmap` encapsules les fonctions nécessaires à la reconstruction de la carte d'environnement, sur GPU, à partir des différentes images issues des cameras ainsi que des données de positionnement correspondantes.

Occlusion : La classe `Occlusion` gère les données relatives aux sphères d'occlusion pré-calculées ainsi que les méthodes de construction des ombres.

Configuration : La classe `Configuration` contient l'ensemble des paramètres lues dans le fichier de configuration.

13. **ZBar :** http://zbar.sourceforge.net/

1.2.2 Modules "videodevice"

La classe virtuelle `VideoDevice` décrit les méthodes utilisées par l'application pour communiquer avec les périphériques d'acquisition. Avec de simplifier leur chargement, les implémentations de cette classes sont encapsulées dans un module qui facilite l'instanciation d'un objet à partir de sources pré-compilées.

L'implémentation des différents prototypes relatives à cette API donne donc le schéma suivant :

Code 1.1 Prototype d'un module `VideoDevice`

```
namespace videodevices
{
  class myVideoObject : public Module<VideoDevice>
  {
    public:
      myVideoObject();
      ~myVideoObject();

      bool       open(int idx = 0);
      void       close();
      bool       isopen();
      void       grabFrame();
      IplImage*  getFrame();
      IplImage*  frame();
      int        getParameter(control);
      void       setParameter(control, int);
      void       resetParameter(control);
      void       showParameters();
    private:
      [...]
  };
};
```

Les méthodes `open`, `close` et `isopen` ont pour rôle de réserver, libérer et indiquer le statut des ressources matérielles. L'entier fournit à la fonction `open` indique quelle camera utiliser. Ces indices commence à 1, l'indice 0 représentant la camera par défaut.

L'acquisition des images, au format `IplImage`, est fait via les fonctions `grabFrame()`, `getFrame()` et `frame()` :

- La méthode `grabFrame()` récupère l'image courante ;
- La méthode `getFrame()` récupère l'image courante et la renvoi ;
- La méthode `frame()` renvoi la dernière image récupérée.

Enfin, les méthodes `getParameter()`, `setParameter()`, `resetParameter()` et `showParameters()` sont utilisées pour le contrôle des paramètres d'exposition et de contraste de la camera.

1.2.3 Modules “scanner”

Les objets correspondant à la classe virtuelle `Scanner` implémentent l'identification des marqueurs présents dans les images fournit par les cameras.

L'implémentation des différents prototypes relatives à cette API donne donc le schéma suivant :

CODE 1.2 Prototype d'un module `Scanner`

```
namespace scanners
{
  class myScannerObject : public Module<Scanner>
  {
    public:
      myScannerObject();
      ~myScannerObject();
      std::vector<Symbol> scan(IplImage*);
    private :
      [...]
  };
};
```

La méthode `scan()` prend une image en paramètre et renvoi la liste des marqueurs identifiés.

Partie 1.3 Configuration

Le comportement de l'application est configurable par le biais d'un fichier de configuration au format XML (voir code **??**, en annexe). Les options sont les suivantes :

OBJECT : Paramètres décrivant l'objet à intégrer à la scène ;

- `obj` : Chemin vers le fichier contenant le maillage (au format `.obj`) ;
- `spheres` : Chemin vers le fichier contenant les sphères englobantes (si cette entrée est vide, une seul sphère, construite à partir de la sphère englobante, sera utilisée) ;
- `scale` : Facteur de mise à l'échelle de l'objet ;

DEVICE : Paramètres décrivant les périphériques à utiliser. `front` représente la camera principale et `back` représente la camera arrière. Pour chacun de ces camera on a les options suivantes ;

- `enable` : Activer / désactiver la camera (désactiver la camera désactive également certains composants pour lesquels elle est indispensable) ;
- `id` : Identifiant de la camera à ouvrir automatiquement ;
- `param` : Chemin vers le fichier de configuration contenant les paramètres de la camera. Si le fichier n'existe pas il sera crée et remplit avec les informations de calibration calculés automatiquement ;

MARKERS : Paramètres des marqueurs ;

- `size` : Taille des marqueur (définit l'unité de distance) ;
- `scale` : Facteur d'échelle entre le marqueur et la mire qui le contient [14] ;

GENERALPARAMETER : Paramètres généraux de l'application

- `verbose` : Niveau de verbosité de l'application ;
- `defaultValues` : Paramètre de luminosité et de contraste. L'option `persistency` indique le nombre d'images pendant lesquels sont conservé les informations de positionnement en cas de perte des marqueurs ;
- `envamp` : Paramètres de l'envmap telle que la définition, l'activation de la construction au démarrage ou le chargement d'une envmap déjà reconstruite ;
- `localisation` : Activer / désactiver les mécanisme de positionnement à partir de marqueurs. Taille de la scène (pour la correction de parallaxe) ;
- `rendering` : Options de rendu (`backgound` affiche l'arrière plan, `scene` rend l'objet et son ombre porté, `view` est un mode de debug) ;
- `modules` : Chemins vers les sources binaires des modules à utiliser.

Partie 1.4 Portage

L'application développée initialement sous Linux, à par la suite été portée sous iOS et est aujourd'hui exécutable sur les périphériques Apple récents. Ce portage à été rendu possible par l'utilisation du C++11, qui est une partie intégrante de l'objective C utilisée pour le développement d'applications iOS.

lesquel version mini- mum ?

De fait, presque aucune modifications de l'application principale n'ont été nécessaires (liées a la non gestion de certaines commandes de l'API OpenGL).

Pour ce qui est des modules, le module de reconnaissance utilisant la bibliothèque multiplate-forme ZBar [15] à pu être re-utilisé en l'état. Seul le module de gestion des cameras à du être récrit pour permettre l'acquisition d'images à partir de l'API propre à iOS.

Les différents shaders ont également du être adapté afin de suivre les spécification OpenGL ES 2.0. OpenGL ES 2.0 étant de fait très proche de OpenGL 3, seul quelques modification, notamment de typage des variables, ont été nécessaires.

14. 1.0 pour un marqueur qui occuperai tout la surface, 0.5 pour un marqueur centré représentant 1/4 de la surface
15. **ZBar :** http://zbar.sourceforge.net/

Bibliographie