PoLAR User's Manual

Pierre-Jean Petitprez, Erwan Kerrien, Pierre-Frédéric Villard

September 29, 2016

1 Introduction

PoLAR (Portable Library for Augmented Reality) is a framework which aims to help creating applications for augmented reality, image visualization and medical imaging. PoLAR is designed to offer useful tools without the need to be a specialist of them. The framework offers the necessary tools to create graphical applications. They are designed to be easy to use and as complete and powerful as possible. The framework is written in C++ and published under the GNU GPL license.
This document aims to explain how to compile PoLAR, how to use PoLAR, and how to add new features to PoLAR.

PoLAR binds two major libraries: **Qt**[1] and **OpenSceneGraph**[2]. Qt is a framework that offers tools for GUI creation, drawing, and a lot of other functionalities, including the management of video streams. OpenSceneGraph is a 3D engine based on OpenGL. It uses scene graphs for the scene representation.

Knowing how to use Qt and OpenSceneGraph is not mandatory to be able to use PoLAR, but it is highly recommended to know at least the basics of Qt to efficiently use PoLAR. Indeed, all the tools in PoLAR are accessible thanks to a custom Qt widget. That way, PoLAR can easily be integrated in any application using Qt widgets. Moreover, PoLAR provides helper functions to use some of Qt and OpenSceneGraph features in an easy way. Qt and OpenSceneGraph features are still reachable in their normal ways. Thereby, beginner users will quickly find everything they need, while advanced users will not feel restricted.

2 Compilation and installation

PoLAR makes use of two major libraries: Qt and OpenSceneGraph. Depending on which platform you are using, the installation of these dependencies is very different and requires more or less work. Linux version of PoLAR is certainly the easiest to install, but we have explained how to make it work on Windows and Mac as well.

2.1 Linux

2.1.1 Dependencies

PoLAR requires some dependencies to be built. On Linux environments, all of them are available as official packages. These packages can be installed on Linux platforms through the package managers. The required packages are listed below:

- Qt packages: `qt5-default qttools5-dev-tools libqt5opengl5-dev qtmultimedia5-dev libqt5multimedia5-plugins qtdeclarative5-dev libqt5svg5-dev`
- OpenSceneGraph packages: `openscenegraph libopenscenegraph-dev`
- OpenGL-related packages: `freeglut3 freeglut3-dev libxmu-dev libxi-dev`

[1] `http://www.qt.io/`
[2] `http://www.openscenegraph.org/`

- For the multimedia functionalities, we recommend using gstreamer:
  `libgstreamer0.10-0 libgstreamer0.10-dev gstreamer-tools gstreamer0.10-plugins-good gnome-media`
- If you want to generate the Doxygen documentation, you will also need these packages:
  `doxygen graphviz`

The minimum required Qt version is version **5.4**. It is available in the default `apt-get` packages on Ubuntu 15.04 and higher. On older versions, you will have to use the source version. The minimum required OpenSceneGraph version is **3.2**, available as packages on Ubuntu 14.04 and higher.

2.1.2 Build

Once the packages are installed, you should not have to change anything in the PoLAR `CMakeLists.txt`. All the packages should be found automatically. If you use source versions, you will have to set the paths to your installation *(see section 2.5)*.
N.B.: you will also need the tools required to generate a `Makefile` from a `CMake` file and a compiler. Recommended tools are **CMake**[3] and **gcc**[4]. Corresponding packages are `cmake` and `build-essential`.

To build PoLAR, create a `build` folder into the PoLAR root directory, and call CMake from it:

```
mkdir build && cd build && cmake ..
```

Then type `make`. The PoLAR library will be compiled in the `lib` folder and the examples will be generated in the `bin` folder.

2.1.3 Installation

You can now install PoLAR in order to use it for external projects. The default installation path is `/usr/local` but you can change this destination by calling the following command from the `build` directory:

```
cmake .. -DCMAKE_INSTALL_PREFIX=<path_to_install_dir>
```

To complete the installation, simply type:

```
make install
```

For the installation to be possible in the default path (`/usr/local`) you might need to use the `sudo` command:

```
sudo make install
```

Note that you can uninstall PoLAR with the following command from the `build` directory:

```
<sudo> make uninstall
```

2.1.4 Using PoLAR in external projects

Once PoLAR is installed, you can use it for your projects. We have provided a file named `FindPoLAR.cmake` located in the `/CMakeModules` directory of PoLAR. Include it in your own CMake-based projects in order to easily find and link PoLAR. To do so, you can create a folder named `CMakeModules` in your project directory and copy the `FindPoLAR.cmake` in this directory. Then, add these lines to your `CMakeLists.txt`:

```
set(CMAKE_MODULE_PATH
 ${CMAKE_MODULE_PATH}
 ${CMAKE_CURRENT_SOURCE_DIR}/CMakeModules)
```

We have provided a template PoLAR project as an example. You can find it in the `/Applications/TemplateApp` folder. It is recommended to start use the `CMakeLists.txt` file of this template project as a basis of your own project.
To compile your project, if PoLAR was not installed to the default location (`/usr/local`), you need to set a particular variable when configuring your project. For example, simply call the following line from the `build` directory of your prjoect:

```
cmake .. -DPoLAR_INSTALL_DIR=<path-to-PoLAR-installation-directory>
```

[3] `https://cmake.org/`
[4] `https://gcc.gnu.org/`

2.2 Windows

On Windows environment, you will have to install all needed libraries and compilation tools first. We recommend using Visual Studio. Visual Studio 2013 has been successfully tested. All versions above VS 2012 should work. Only 32 bit compilation was successfully tested. You will also need to install the Windows SDK[5].

2.2.1 Qt

We recommend using the online Qt installer[6]. The minimal required version is Qt 5.4. Precompiled binaries for Visual Studio are provided.
To install Qt, run the installer you have previously downloaded. Skip the part asking for personal information. Then choose the path to install Qt. We recommend to install Qt into the `C:\Qt` folder, to help PoLAR easily finding it. Then choose the packages to install. Be sure to choose the one corresponding to your Visual Studio version. For example, choose **Qt 5.5, msvc2013 32-bit** for Visual Studio 2013 32-bit. Then run the installation.

2.2.2 OpenSceneGraph

Due to the lack of recent binaries, we recommend compiling from sources[7]. Download the latest stable release as a ZIP archive. Extract this archive. We recommend to extract it in the `C:\` directory, so that PoLAR will easily find it.

Third party dependencies
You may need to add some third party libraries (e.g. for loading some particular file formats). The official OpenSceneGraph website[8] provides Visual Studio prebuilt dependencies. In the case you would prefer installing the dependencies from their sources, please refer to appendix A.

To use the prebuilt dependencies, proceed as follows:

- Download the dependencies that match your Visual Stduio version from the official webpage. Beware, the packages are heavy (more than 2 Go when extracted).
- Some of the dependencies (for example the Visual Studio 2013 ones) contain two folders: `x86` and `x64`, corresponding respectively to a 32-bit platform and to a 64-bit platform. We have tested only a 32-bit compilation.
- Extract the folder corresponding to your platform into your OpenSceneGraph root directory (for example `C:\OpenSceneGraph-3.4.0`) and rename it `3rdParty`.

Compiling OpenSceneGraph

- Create a folder named `build` inside the OpenSceneGraph root directory. Run CMake; choose the OpenSceneGraph root directory (`C:\OpenSceneGraph-3.x.x`) as source code entry, and the `build` folder previously created as build directory. Click on **Configure**.
- If you have added some dependencies using appendix A, you will need to add the path to them. Search the corresponding variables in the CMake output (for example, for the JPEG dependency, the 2 interesting variables are `JPEG_INCLUDE_DIR` and `JPEG_LIBRARY`) and modifiy their value to match with the path to the dependencies. For example, set:
`JPEG_INCLUDE_DIR` to `C:\OpenSceneGraph-3.4.0\osg-3rdparty-cmake-master\libjpeg`
and
`JPEG_LIBRARY` to `C:\OpenSceneGraph-3.4.0\osg-3rdparty-cmake-master\lib\Release\jpeg.lib`
if you have followed our recommendations.
- If you have added prebuilt dependencies downloaded from the official webpage, set the CMake variable `ACTUAL_3RD_PARTY_DIR` to the folder you have extracted (for example, following our recommendations from the previous section: `C:/OpenSceneGraph-3.4.0/3rdParty`).

[5] https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk
[6] http://www.qt.io/download-open-source/
[7] http://www.openscenegraph.org/index.php/download-section/stable-releases
[8] http://www.openscenegraph.org/index.php/download-section/32-third-party

Note: in some cases, using backslashes ("\") for paths will lead to a parsing error. It is recommended to use regular slashes ("/") for Windows paths.

- Hit `Configure` again; when done, hit `Generate`. It will generate the Visual Studio solution. Run Visual Studio and load the OpenSceneGraph solution that can be found in the `build` directory. You only need to hit F7 to launch the project compilation and everything should be fine. Be sure to compile OpenSceneGraph as `RELEASE` if you want to use PoLAR in `Release` mode. Note that the compilation can take a very long time.

When the compilation is finished, in order to be able to use OpenSceneGraph, you will have to add the path to the OpenSceneGraph libraries into the Windows path. To do so:

- Go to the Windows configuration panel. Launch the **System** properties panel. Click on the **Environment variables** button.
- A new window appears. In the bottom part, find the line corresponding to the variable named **Path**. Select it and click on **Modify**.
- In the new popup window, add these values in the **value** field:
  `C:\OpenSceneGraph-3.4.0\build\bin;`
  `C:\OpenSceneGraph-3.4.0\build\bin\osgPlugins-3.4.0;`
  If you installed OpenSceneGraph in another directory, please modify these lines accordingly. Modify the OpenSceneGraph version number accordingly too. **Do not remove any already existing value**; add the new ones at the beginning of the line.
- Validate and close the configuration panel.

Compiling PoLAR

When Qt and OpenSceneGraph are compiled and installed, you will need to add references to their installation folders in the PoLAR `CMakeLists.txt`. We recommend using CMake to generate a Visual Studio project from this `CMakeLists.txt`. If you have followed our recommandations for installing Qt and OpenSceneGraph, PoLAR should automatically find them. Otherwise, you will need to set some CMake variables. Please refer to section 2.5 for the useful CMake options.
Create a `build` directory inside the PoLAR directory and set it as build folder in CMake. Then click on `Configure` ans then on `Generate`. In Visual Studio, be sure to check if the right mode (`RELEASE` or `DEBUG`) is selected. Then, you just have to run the compilation.

Linking PoLAR to external projects

Once PoLAR is compiled, you have to add the path to the library in the Windows path, just like for OpenSceneGraph. Following the same steps, add the path `<Path-to-PoLAR-build-dir>\bin\Release` to the Windows path.

To use PoLAR in an external project, you can take the project named `TemplateApp` in the `Applications` folder from the PoLAR root directory. With CMake, create a Visual Studio project. The configuration will fail the first time you run it. Look for the CMake variables `PoLAR_INCLUDE_DIR` and `PoLAR_LIBRARY_DIR`, and set values to them, respectively `<Path-to-PoLAR-root-dir>\include` and `<Path-to-PoLAR-build-dir>\lib\Release`. Now the configuration and the generation should work. In Visual Studio, be sure to check if the right mode (`RELEASE` or `DEBUG`) is selected. Then, you just have to run the compilation.
Note: if you want to use the debug version of PoLAR, just replace `Release` by `Debug` in the above paths.

2.3 Mac OS

If not already done, install X-Code, which is mandatory for most of development work in Mac OS. It is available through the App Store. Once installed you will need to run two extra command lines in a Terminal:

- `sudo xcode-select -s /Applications/Xcode.app/Contents/Developer`

- `sudo xcodebuild -license` and agree with the licence

PoLAR requires some dependencies to be built. On Mac OS environments, all of them are available as packages. We recommend to use Homebrew[9] that has been successfully tested with PoLAR. To install Homebrew simply run:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

The required packages are listed below:

- All the basic tools to download the project and to compile it:
  `brew install git cmake`
- OpenSceneGraph packages version **3.2** exactly:
  `brew install -vd https://github.com/Homebrew/homebrew/commit/bd342884aacb198003be479ba08feacdcc82d250#Library/Formula/open-scene-graph.rb`
  It will also install **QT 5.5** by default. You will need to add the library path (for instance, in `/.profile`):
  `export DYLD_LIBRARY_PATH= /usr/local/Cellar/open-scene-graph/3.2.0/lib/:$DYLD_LIBRARY_PATH`
- If you want to generate the Doxygen documentation, you will also need these packages:
  `brew install doxygen graphviz`

Once the packages are installed, you should not have to change anything in the PoLAR `CMakeLists.txt`. All the packages should be found automatically. If you use source versions instead, you will have to set the paths to your installation *(see section 2.5)*.
To build PoLAR, create a `build` folder into the PoLAR root directory, and call CMake from it:

```
mkdir build && cd build && cmake ..
```

Then type `make`. The PoLAR library will be compiled in the `lib` folder and the examples will be generated in the `bin` folder.

This installation protocol has been tested on OS X El Capitan (version 10.11.3).

2.4 Android

In order to compile PoLAR and use it on Android, you will need to follow some preliminar step:

- First, follow the official Qt recommendation[10] for installing the required tools (Android SDK, NDK, JDK). You will also need to install `ant` if not already installed. Note for the SDK: download the command line tools only. Note for the JDK: the minimum version required is JDK 7.
- Once everything is installed, download the Qt binaries for Android. You can choose any version above Qt 5.4. We recommend using the Qt online installer. It will also install a proper IDE (Qt Creator), needed to compile the application on the target platform.
- You will need to compile OpenSceneGraph for Android from the sources[11]. You can add the third party dependencies from here: Don't forget to set an installation directory and to run `make install`. The path to the installation folder will be used by PoLAR. You can follow the steps explained in the official OSG guide[12]. To be compliant with PoLAR, you need to choose "GLES2" as OpenGL profile.

Now you can compile PoLAR. You can follow the same steps as for OpenSceneGraph: create a folder in the PoLAR root directory, for exemple `/build-android`. From this folder, run

[9] `http://brew.sh`
[10] `http://doc.qt.io/qt-5/androidgs.html`
[11] `http://www.openscenegraph.org/index.php/download-section/stable-releases`
[12] `http://www.openscenegraph.org/index.php/documentation/platform-specifics/android/178-building-openscenegraph-for-android-3-4`

```
cmake -DANDROID_NDK=<path-to-NDK>
-DCMAKE_TOOLCHAIN_FILE="../PlatformSpecifics/Android/android.toolchain.cmake"
-DCMAKE_BUILD_TYPE=Release -DANDROID_ABI="armeabi-v7a" -DANDROID_NATIVE_API_LEVEL=19
-DOSG_CUSTOM_DIR=<path-to-OSG> -DQT_CUSTOM_DIR=<path_to_Qt>
-DCMAKE_INSTALL_PREFIX=<path-to-the-install-path> ..
```

where ANDROID_ABI is the target hardware and DANDROID_NATIVE_API_LEVEL is the target Android API level (for example 19 corresponds to Android 4.4 KitKat).

CMake variables QT_CUSTOM_DIR and OSG_CUSTOM_DIR are the paths to the custom Qt and OpenSceneGraph installation directories. Note that in the case of Qt, you must set the path to the subdirectory for the desired version and not to the root directory (for example, if you have installed Qt in your home directory, the path for QT_CUSTOM_DIR should be /home/login/Qt/5.7/android_armv7 and not just /home/login/Qt). Then run make.
Now you can install PoLAR. You can follow the steps from section 2.1.3. We recommend installing PoLAR in *a different folder* from a regular non-Android PoLAR installation. For example, create a folder named PoLAR-Android and use it as installation target.

Now it is time to compile an example using PoLAR on Android: look at the example runAndroid in the /Applications/runAndroid directory. Open the project runAndroid.pro with Qt Creator. Now you need to configure Qt Creator in order to be able to compile and deploy the Android application:

- Goto Tools->Options->Android and, if Qt Creator has not found them automatically, set the paths to your JDK, your Android SDK, your Android NDK and Ant.
- Then goto the Projects tab and open up the Build environment tab. Check that the variable ANDROID_NDK_PLATFORM corresponds to the same target platform than the one used to compile OpenSneneGraph and PoLAR. For example, if you compiled OpenSceneGraph and PoLAR with ANDROID_NATIVE_API_LEVEL=19, the variable ANDROID_NDK_PLATFORM must be set to android-19.
- Change the paths to PoLAR and to OpenSceneGraph in the pro file of the project to the ones corresponding to your own installation. Change also the OpenSceneGraph version (OSG_VERSION varialbe in the same .pro file) to the one you are using.
- You can now connect your device to your computer. Compile and deploy the application by clicking on the green arrow at the bottom left. Choose your target device and everything should go fine.

Note: In order for the runAndroid example to run with the correct images and 3D models, you must copy the data folder provided in the PoLAR Examples directory to your device. The example seeks for the folder in /sdcard/polar/data/. Create a polar directory at the root of your device and paste the data folder in it.

2.5 CMakeFile options

PoLAR uses CMake for the build steps. Use the way you prefer to generate the Makefile from the CMakeLists.txt file. For platform specific details, see the previous sections.
In the CMakeLists file, some options can be modified:

- BUILD_PoLAR_EXAMPLES: set it to ON if you want to compile the examples provided with PoLAR. Thoses examples show the different features of PoLAR.
- QT_CUSTOM_DIR, OSG_CUSTOM_DIR: require a path in string format to a Qt or OSG custom installation folder. When set, CMake will use these paths to find Qt and OpenSceneGraph in the case packaged versions are not found (e.g. on Windows).
- USE_FIXED_PIPELINE: on desktop enviroments, this variable is set to OFF by default. On mobile platforms this variable is set to ON since no fixed-function pipeline is available. You can set it to ON to disable usage of fixed-function pipeline.
- USE_PHONG_SHADING: default is OFF on desktop, ON on Android. Set it to ON to use Phong shading shaders instead of fixed-function shading.

- USE_SHADOWS: set to ON by default, you can set it to OFF to completely disable shadow management.
- USE_DEPTH_MAPS: default is ON on desktop, OFF on Android. Enables the use of depth maps for shadow generation. On mobile platforms, depth maps require an extension which is not always available.
- USE_OSGSHADOW: default is ON on desktop, OFF on Android. Set it to ON to use the OpenSceneGraph shadows (unavailable on Android). Set it to OFF to use PoLAR custom shadow management (requires USE_FIXED_PIPELINE to be OFF and USE_PHONG_SHADING to be ON).
- DYNAMIC_PoLAR: if ON, PoLAR will be built for dynamic linking. If OFF, PoLAR will be built for static linking.
- USE_OFF_PLUGIN: if ON, the plugin for loading OFF 3D models in OpenSceneGraph will be compiled. It will be placed in the same output folder than the PoLAR library.
- OSG_USE_FBO: default to ON. Setting it to OFF will deactivate the use of a custom version of `osg::CullVisitor`, used to fix an incompatibility between Qt and OSG regarding the use of FBOs.
- SET_LOCALE_NUMERIC_C: set it to ON to set the locale numeric configuration to C. Set it to OFF if you want to apply your own locale numeric configuration.
- DOXYGEN_GENERATION: if set to ON, the doxygen documentation will be generated and will be accessible in the Doc folder.

2.6 Running the examples

We provide some examples to show the different possibilities of PoLAR. The sources are in the /Examples subdirectory. In this folder you will also find a readme file describing every example and how to use them. These examples show almost all the features that are available in PoLAR. Some examples show different ways to include a `PoLAR::Viewer` in a Qt application, and the techniques to inherit from a `PoLAR::Viewer`.
With the default installation, the binaries will be found in the /bin folder. We also provide some test assets (3D models, textures, images, projection matrices, etc.) in the /Examples/data folder.

Most examples require command-line options to work. You can find proper options which use the provided test assets by refering to the README.txt file in the /Examples folder.

On Windows, you will need to help the binaries find the Qt and OpenSceneGraph libraries. The simplest way is to add their path in the PATH environment variable. Please refer to the Windows documentation for more details.

As often as possible, we have written which example you can refer to for each example studied in this document.

3 Using PoLAR

3.1 Getting started

PoLAR is designed to provide the most useful features when creating a graphical application in the fields of augmented reality and medical imaging.
When using PoLAR, it is best to keep in memory that its first usage is as a graphical library. Some features do not need any graphical context or window to be accessible, but the main functionalities are implemented around a Qt widget. In other words these functionalities can only be used after having created a custom PoLAR widget (`PoLAR::Viewer`).
The `PoLAR::Viewer` manages everything needed for a graphical application. You do not need to understand how Qt and OpenSceneGraph work (even if it is recommended): PoLAR does everything for you.

3.1.1 Hello world

The code above will create a PoLAR::Viewer (and thus a Qt application using PoLAR) in the simplest possible way.

```
#include <QtWidgets/QApplication> // for Qt functions
#include <PoLAR/Viewer.h> // PoLAR::Viewer header

int main(int argc,char ** argv)
{
   // Create the Qt application
   QApplication app(argc, argv);

   // Create the Qt widget, with no parent and a name
   PoLAR::Viewer viewer;

   // Resize the widget
   int width = 640;
   int height = 480;
   viewer.resize(width, height);

   // Show the widget
   viewer.show();

   // This line ensures the application will properly close when the widget has been closed
   app.connect( &app, SIGNAL(lastWindowClosed()), &app, SLOT(quit()) );

   // Run the app
   return app.exec();
}
```

The result is shown on figure 1.

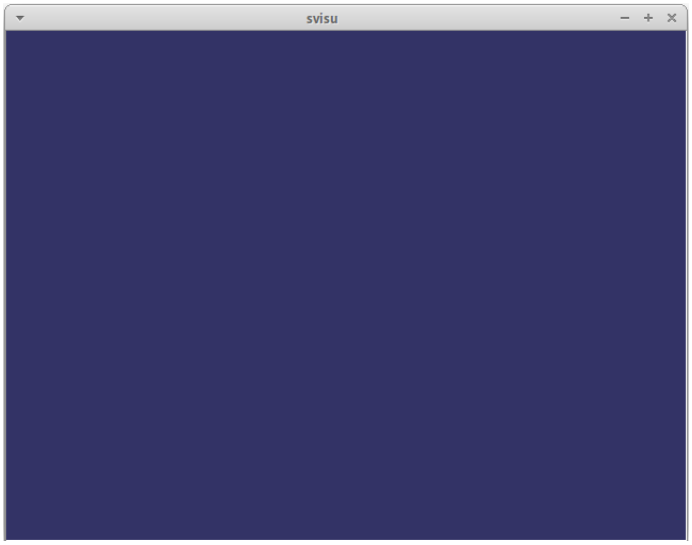

Figure 1: A simple PoLAR::Viewer

Of course this is not very useful for the moment. We will see further in this document how to develop real applications with PoLAR.

3.1.2 Interface modes

While some functionalities are always available in any circumstances, PoLAR is built around different *modes*. These modes are different classes of features and allow to separate the interaction methods with the application. Currently, PoLAR is divided into three modes:

- Image edition mode: allows manipulating the image displayed as background *(see section 3.2)*
- 2D objects mode: enables the posibility to draw geometrical shapes on the viewer *(see section 3.3)*

- 3D edition mode: enables the manipulation of the 3D scene *(see section 3.4)*

Note that enabling one mode disables the other ones. This way, the same key bindings can be used in the different modes, without the *multiple usage for the same key* problems.
The currently active mode can be found thanks to some helper methods:

```
PoLAR::Viewer::EditionMode mode = viewer.getEditionMode();
bool a = viewer.isInImageEditionMode();
bool b = viewer.isIn2DObjectsEditionMode();
bool c = viewer.isIn3DSceneEditionMode();
```

The next parts contain more detailed descriptions of these different modes.

3.2 Image viewer

The simplest way to use PoLAR is as an image viewer. Creating a `PoLAR::Viewer` will create all the tools needed for basic image manipulation: image loading from file or from buffer, panning, zooming, and adjustment of window/level (brightness and contrast following a look-up table).

3.2.1 Static images

PoLAR provides an image type adapted to the needs of medical imaging and augmented reality. The `PoLAR::Image` class is a templated class which allows to manage data in different numeric types (`float, double, int, char`, etc). A basic image which is only used as a texture can be stored as `unsigned char` (RGB: 3 bytes per pixel / grayscale: 1 byte per pixel), which corresponds to a classic image format. More complex data on which it can be needed to apply some algorithms will more likely be stored as `float` or even `double`.

`PoLAR::Image` inherits from `osg::Image`. A `PoLAR::Image` can then be used as a texture on any 3D object in the scene. `PoLAR::Image` is also used as background image of the `PoLAR::Viewer`. The background image is not linked to the 3D scene. It is always displayed in an orthographic view and is used to display textures as background of the scene or simply as background of the viewer.

To create a `PoLAR::Image`, it is possible to set to the constructor either an `osg::Image`, or a file name (to a texture stored on the hard drive), or a pointer to a buffer (of type `float, double, char, int,` ... The created templated `PoLAR::Image` will be of the same internal type).

Let's take our example from figure 1. We will add a custom `PoLAR::Image` as a background. Please refer to the example named `svisu` in the `Examples` folder.
First, we include the correct header:

```
#include <PoLAR/Image.h>
```

Now, we can load a `PoLAR::Image` from an image file:

```
//Using OSG smart pointers
osg::ref_ptr<PoLAR::Image_uc> myImage;
...
// Open the image
myImage = new PoLAR::Image_uc("myImage.png", true);

// Resize the widget according to the size of the image read
width = myImage->getWidth();
height = myImage->getHeight();
viewer.resize(width, height);

// Add the image read as background image
viewer.setBgImage(myImage);

// Show it
viewer.bgImageOn();

// activate the image manipulation mode
viewer.startEditImageSlot();
...
```

The result is shown on figure 2.

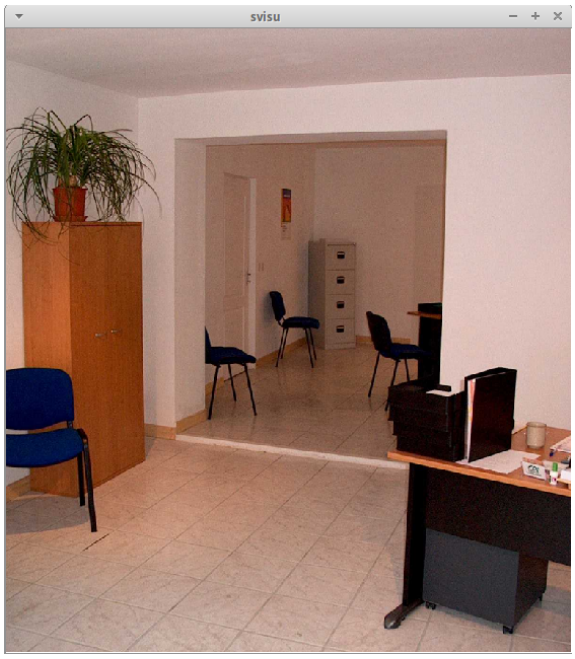

Figure 2: A simple PoLAR::Viewer with an image in background

Note the use of a *smart pointer* to store the image. Its goal is to ease the memory management: a resource dynamically allocated with the `new` keyword is automatically deleted when the last smart pointer that points to it is deleted. PoLAR makes use of the `osg::ref_ptr` type provided by OpenSceneGraph for all its internal resources.

Let's just focus on the constructor of the `PoLAR::Image` class:

```
myImage = new PoLAR::Image_uc("myImage.png", true);
```

This line is equivalent to

```
myImage = new PoLAR::Image<unsigned char> ("myImage.png", true);
```

Indeed, as said before, `PoLAR::Image` is a templated class. In the case of a bitmap image, using `unsigned char` is the best choice. We have created an alias for the use of `unsigned char` since it is the most common usage.
Now let's focus on the arguments. The first argument is the path to the image file. `PoLAR::Image` uses the OpenSceneGraph `osgDB::readNodeFile()` method to open the images: all the formats known by OpenSceneGraph can be opened as `PoLAR::Image`.
The second argument is a boolean and is optional. Its default value is `false`. If set to `true`, the image will be flipped vertically. This is to ensure the image is displayed in the correct way because in some circumstances, OpenSceneGraph flips the image vertically at loading.

3.2.2 Video stream

PoLAR is able to display two kinds of video media:

- **video stream** from a camera connected to the computer (webcam)
- **from a recorded file** on the hard drive (as a `.mp4` file for example)

This feature uses the Qt classes `QCamera` and `QMediaPlayer`. In the case of a recorded media, basic features like rewind/forward, play, pause, stop, or go at a given timestamp, are implemented.

By linking the newly created `PoLAR::VideoPlayer` to a `PoLAR::Image`, you can use your video stream as a texture in your scene, in the same way as you would do with a static image. That means

you can create dynamic textures and apply them on 3D objects.

The ways of creating a `PoLAR::VideoPlayer` for using a webcam or for playing a recorded file are very similar. You can refer to the examples `svisucam` and `svisuvideo`.

Using a webcam:

```
#include <PoLAR/Image.h>
#include <PoLAR/VideoPlayer.h>

...
osg::ref_ptr<PoLAR::Image<unsigned char> >
    myImage;
osg::ref_ptr<PoLAR::VideoPlayer> camera;
// the device number: /dev/videoX
int camNumber = 0;

camera = new PoLAR::VideoPlayer(camNumber);
myImage = new PoLAR::Image
      <unsigned char>(camera.get());

viewer.setBgImage(myImage.get());
...
```

Playing a media file:

```
#include "Image.h"
#include "VideoPlayer.h"

...
osg::ref_ptr<PoLAR::Image<unsigned char> >
    myImage;
osg::ref_ptr<PoLAR::VideoPlayer> camera;
// path to the media file
char* filename = "myFile.ogv";

camera = new PoLAR::VideoPlayer(filename);
myImage = new PoLAR::Image
    <unsigned char>(camera.get());

viewer.setBgImage(myImage.get());
...
```

The only difference is in the `PoLAR::VideoPlayer` constructor. In the case of a webcam, the first parameter passed to the constructor must be the device number, i.e. the number obtained when looking into the `/dev` folder (on Linux environments). For example, the first webcam connected to a computer will be #0: it will appear as `/dev/video0`.
In the case of a media file, the first parameter has to be the path to the media file (in absolute or relative form).

N.B.: in the code above, only one parameter is passed to the `PoLAR::VideoPlayer` constructor. In this case, the width and height of the frames coming from the webcam or from the media file will be stored and displayed at their real size. For example, for a webcam displaying frames in 640x480px the linked `PoLAR::Image` will be sized to 640x480px too. As a side effect, when using this image as background image, the `PoLAR::Viewer` displaying this image will also be scaled to fit this size.

It is however possible to force the framework to work with custom width and height. To do that, give your custom width and height as second and third parameters in the `PoLAR::VideoPlayer` constructor:

```
camera = new PoLAR::VideoPlayer(camNumber, myWidth, myHeight);
```

or

```
camera = new PoLAR::VideoPlayer(filename, myWidth, myHeight);
```

Note however that such forced scaling might possibly lead to a slow-down of the framerate.

3.2.3 Common interaction modes

Let's focus on this interesting line:

```
// activate the image manipulation mode
viewer.startEditImageSlot();
```

This method is a *slot*. *Slots* and *signals* are a big part of the Qt programming style. For more information about them, please consult the Qt documentation[13]. Slots can be used as normal methods. It is the case here. This method activates the *image edition* mode, as described in part 3.1.2.
Thanks to this mode, we can now interact with the image displayed in background of the viewer. The possible manipulations are:

- Pan: move the image along the X and Y directions. The default interaction method for this manipulation is using the left mouse button: click and drag along X or Y for panning.

[13] `http://doc.qt.io/qt-5/signalsandslots.html`

- Zoom: zoom in or out on the image. the default interaction method for this manipulation is the right mouse button: click and drag along Y for zooming.
- Window/level: change the brightness and contrast following a *lookup table* (LUT). The default interaction method for this manipulation is the mouse wheel (or mouse middle button): click and drag to change the window/level.

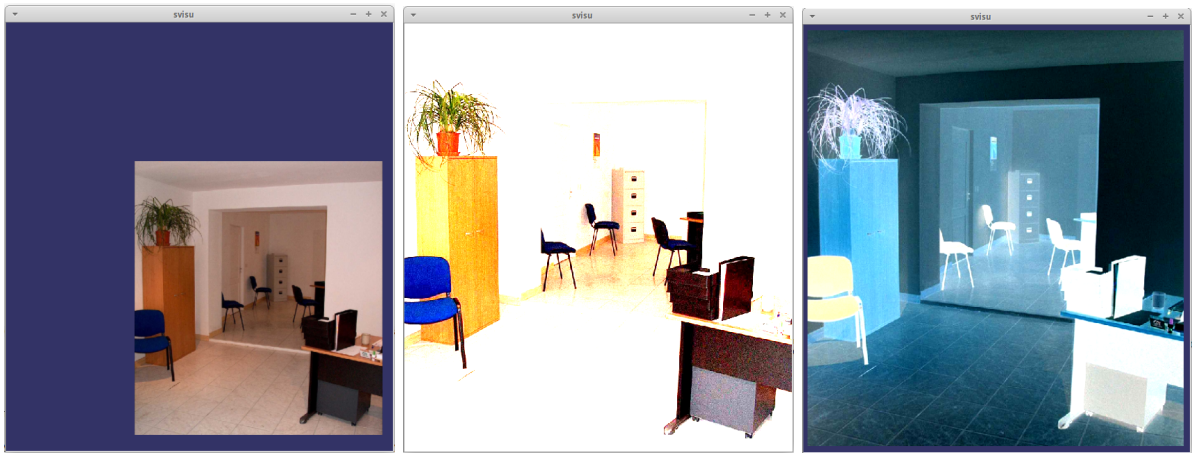

Figure 3: Different values of pan, zoom and window/level

The default interaction methods are customizable thanks to these methods:

```
PoLAR::Viewer::setPanMouseButton(Qt::MouseButton button, Qt::KeyboardModifiers modifier);
PoLAR::Viewer::setZoomMouseButton(Qt::MouseButton button, Qt::KeyboardModifiers modifier);
PoLAR::Viewer::setWindowLevelMouseButton(Qt::MouseButton button, Qt::KeyboardModifiers
    modifier);
```

The `Qt::KeyboardModifiers modifier` parameter is facultative. Its default value is `Qt::NoModifier` (no modifier applied). The other possible values are `Qt::ShiftModifier`, `Qt::ControlModifier`, and `Qt::AltModifier`. Note that Qt does not make any difference between right and left keys.

Moreover, in the case of the zoom, it is possible to choose between dragging with a mouse button clicked, or scrolling with the mouse wheel:

```
PoLAR::Viewer::setZoomMode(PoLAR::PanZoom::ZoomMode mode, float precision);
```

`PanZoom::ZoomMode` has two values: `DRAG` or `SCROLL`. The precision is a facultative parameter whose default value is 1. Depending on the resolution of the mouse wheel, values equal to or even higher than 10 can be necessary to get a smooth effect:

```
viewer.setZoomMode(PoLAR::PanZoom::SCROLL, 10.0f);
```

3.3 2D objects

3.3.1 Overview

PoLAR offers a toolbox for 2D drawings, allowing to draw some control points and some basic geometrical shapes on the viewer.

2D objects are implemented using two different classes. They represent respectively the mathematical representation and its graphical representation. Their class names are listed in the following list:

Mathematical representation	Graphical representation	Purpose
Object2D	DrawableObject2D	Abstract type (not instanciable)
Markers2D	DrawableMarkers2D	Simple control points whose shape is modifiable: square, cross, circle, or invisible
Spline2D	DrawableSpline2D	Spline. The graphical representation follows a Bezier curve
Blob2D	DrawableBlob2D	Closed spline
Polygon2D	DrawablePolygon2D	Closed polygon
PolyLine2D	DrawablePolyLine2D	Open polyline
Arrows2D	DrawableArrows2D	Arrows, i.e. a segment between two markers with the possibility to draw an arrow shape at one end
Text2D	DrawableText2D	Text displayed next to a control point

The different 2D objects are shown on figure 4.

There are different ways for creating 2D objects. The easiest way is to use the helper methods provided in the viewer's API.

```
viewer.newMarkers();
viewer.newSpline();
viewer.newBlob();
viewer.newPolygon();
viewer.newPolyLine();
viewer.newArrows();
viewer.newText();
viewer.newText(QString text);
```

The only method that can take a parameter is `newText(QString text)`. That way, the displayed text will be the one set in parameter. Using `newText()` without parameter creates a Text object with an empty text that can be replaced later.

These methods call `startEditMarkersSlot()` in the `PoLAR::Viewer`. The interaction mode is thus automatically switched to `OBJECT2D` mode. Now, you can add and delete markers (control points) in the selected object, and select and delete 2D objects (see section 3.3.3 for more information).

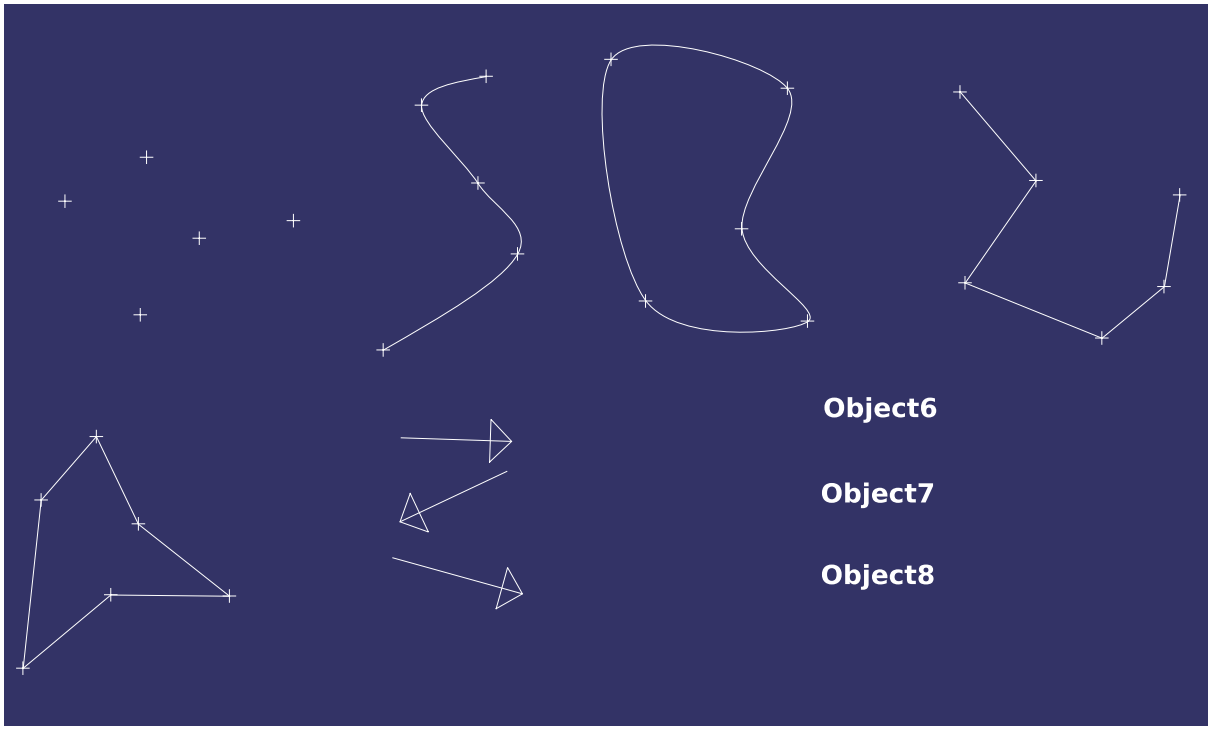

Figure 4: The different 2D objects available
From top left to bottom right: markers, spline, blob, polyline, polygon, arrows, text

Please refer to the example `demo2D` for a complete example on how to create and use 2D objects.

3.3.2 Customizing 2D objects

2D objects are fully customizable: the shapes of their control points, their color (when selected, when unselected, when one control point is selected), if they are displayable or not, etc. All these specifications have a default value:

- **Control points:** a square when the object is selected, a cross when unselected
- **Colors:**
  - Paths: yellow when selected, blue when unselected
  - Control points: red for the selected one. The other ones take the path's color.

Specific features:

- **Arrows2D:** control points are displayed, arrow-shaped control point is not active (i.e. the shape is like a simple segment)
- **Text2D:** control points are undisplayed.

You can also use the Qt `QPen`. For each different category (selected path, unselected path, selected marker in the currently selected path), you can set a custom `QPen` that will be used for the drawing:

```
PoLAR::DrawableObject2D::setPen(QPen pen, PenType type);
```

with `PenType` being either `Selected`, `Unselected`, or `MarkerSelected`.

Instead of setting the `QPen` objects one by one, you also can set the three different ones with one method:

```
PoLAR::DrawableObject2D::setPens(QPen select, QPen unselect, QPen selectMarker);
```

Or you can also set them in the constructor:

```
DrawableObject2D(Viewer2D *Parent, QPen unselectPen, QPen selectPen, QPen selectMarkerPen);
```

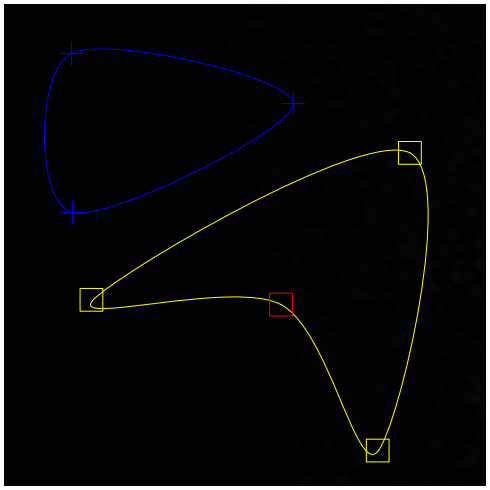

Figure 5: Default colors and control point shapes
In blue: unselected object; in yellow: selected object; in red: selected control point

3.3.3 2D objects manipulation

The manipulation of 2D objects is intended to be as simple as possible. Default commands are:

- **mouse left button**: select (if clicked and released) or move (is clicked and dragged) a control point of the currently active object
- **mouse middle button**: create or select a control point of the currently active object. If the control point is already selected, using again the same button prints out its coordinates.
- **mouse right button**: delete a control point in the currently active object
- **shift + mouse left button**: select (activate) an object among all the currently displayed objects
- **shift + right left button**: delete the object. It must already be active (selected) before deleting it, otherwise the command has no effect.

These are the default commands, but all of them can be modified. For each of the above actions, a mouse button (left/middle/right) and a modifier (shift/control/alt or no modifier) can be set.
To change the commands, use these methods:

```
PoLAR::Viewer::setSelectObject2DManipulation(Qt::KeyboardModifier m, Qt::MouseButton b);
PoLAR::Viewer::setDeleteObject2DManipulation(Qt::KeyboardModifier m, Qt::MouseButton b);
PoLAR::Viewer::setAddMarker2DManipulation(Qt::KeyboardModifier m, Qt::MouseButton b);
PoLAR::Viewer::setSelectMarker2DManipulation(Qt::KeyboardModifier m, Qt::MouseButton b);
PoLAR::Viewer::setDeleteMarker2DManipulation(Qt::KeyboardModifier m, Qt::MouseButton b);
```

The `Qt::KeyboardModifier modifier` parameter means Control, Atl, Shift or none.
Note that custom commands are not checked. If you unintentionally set two indentical commands to two different actions, no security system will warn you.

3.4 3D scenes

3.4.1 Overview

PoLAR manages 3D scenes thanks to the OpenSceneGraph 3D engine, taking advantage of its scene graph representation.
The provided features are:

- addition/deletion of 3D objects at runtime
- object manipulation: position, scale, etc.
- scene/camera manipulation
- shadows and materials, texturing, lighting. See the `osg::Material` class for more information.
- projection matrices **(matrix computation not included into PoLAR)**

PoLAR provides its own class of 3D objects: `PoLAR::Object3D`. This format encapsulates an `osg::Node`, i.e. an OpenSceneGraph scene node, and adds many helper methods to use them.

3.4.2 Adding objects to the scene graph

As for the 2D objects, once a PoLAR viewer is created, everything needed for creating 3D scenes is set. The viewer provides the API needed to interact with the scene graph. More advanced users can still have direct access to the scene graph, but it is generally not necessary.
PoLAR provides a high-level API to interact with the OpenSceneGraph scene graph, encapsulated into the `PoLAR::SceneGraph` class. It is generally not necessary to directly call this class, since most useful features can be accessed from the viewer's API.

The scene graph in a PoLAR application is built following this hierarchy:

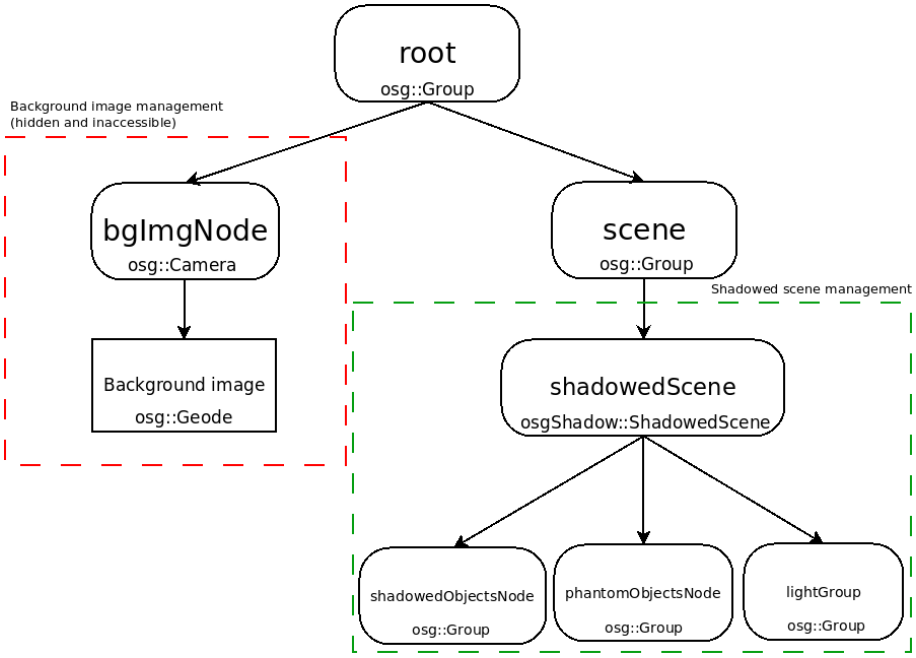

Figure 6: Scene graph hierarchy in PoLAR

The important things to keep in mind are:

- The **shadowedObjectsNode** node will be parent of all classic 3D objects in the scene;
- The **phantomObjectsNode** node will be parent of all phantom objects in the scene (see section 3.4.5);

- The **lightGroup** node will be parent of all lights in the scene;
- The **shadowedScene** node is either a `osgShadow`[14] node, either a simple `osg::Group` (depending on if the CMake variable `USE_OSGSHADOW` has been set to ON or OFF), which manages the shadows in the scene. In the case of an `osgshadow` node, the default shadow system uses a soft shadow map. It is however possible to provide another `osgShadow` shadows technique. See section 3.4.5 for more details.

Advanced users who would like to access the scene graph can call it following this method:

```
#include <PoLAR/Viewer.h>
#include <PoLAR/SceneGraph.h>

PoLAR::Viewer viewer;
...
PoLAR::SceneGraph * sceneGraph = viewer.getSceneGraph();

// access to the root node of the scene graph
osg::Group* root = sceneGraph->getSceneGraph();

// access to the scene node of the scene graph
osg::Group* scene = sceneGraph->getScene();

// access to the shadwoed scene node of the scene graph
osgShadow::ShadowedScene* shadowedScene = sceneGraph->getShadowedObjectsNode();

// access to the objects node of the scene graph
osg::Group* objGroup = sceneGraph->getShadowedObjectsNode();

// access to the phantom objects node of the scene graph
osg::Group* phantoms = sceneGraph->getPhantomObjectsNode();

// access to the objects node of the scene graph
osg::Group* lights = sceneGraph->getLightGroup();
```

Loading 3D objects

Generally, 3D objects will have to be loaded from a file. To do so, the easiest way is to set the name as a parameter of the `PoLAR::Object3D` constructor:

```
#include <PoLAR/Object3D.h>

...
const char[] objName = "myObject.obj";
osg::ref_ptr<PoLAR::Object3D> obj = new PoLAR::Object3D(objName);
```

But it is also possible to encapsulate an already existing 3D node into a `PoLAR::Object3D`:

```
#include <osgDB/ReadFile>
#include <PoLAR/Object3D.h>

...
osg::Node* node;
...
osg::ref_ptr<PoLAR::Object3D> obj = new PoLAR::Object3D(node);
```

Note: PoLAR makes use of OpenSceneGraph *smart pointers*. They are meant to avoid tedious memory management, particularly in complicated cases where a same object can be shared between different parents. When manipulating 3D objects, it is best to use these smart pointers. These smart pointers are represented by a special class called `osg::ref_ptr`. Please refer to the OpenSceneGraph documentation[15] for more details on the `osg::ref_ptr` class.

When creating a `PoLAR::Object3D`, some arguments can be set to the constructors:

[14] `http://trac.openscenegraph.org/projects/osg//wiki/Support/ProgrammingGuide/osgShadow`
[15] `http://trac.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/a00666.html`

```
// constructor from a file
PoLAR::Object3D(char *name, bool material, bool editable, bool pickable);
// contructor from an already existing node
PoLAR::Object3D(osg::Node *node, bool material, bool editable, bool pickable);
```

The `node` is the `osg::Node` containing the geometry. The three booleans are facultative, their default values are `false`.

- `material`: if set to true, a default material will be created for the 3D object. If the node already has one, its material will not be overriden.
- `editable`: if set to true, the object's position, rotation and scale can be programmatically changed, using these different methods:

  ```
  osg::ref_ptr<PoLAR::Object3D> myObject;
  ...
  myObject->translate(1.0f, 1.0f, 1.0f); // you can also pass a osg::Vec3 vector
  myObject->scale(2.0f); // you can also set a different scale factor for X, Y and Z
  myObject->rotate(M_PI/2, 1, 0, 0); // you can also pass a osg::Quat quaternion
  ```

  Other methods are available, particularly some Qt *slots* providing the possibility to connect them to actions from a GUI. Note that you can always change the `editable` option later with the methods

  ```
  myObject->setEditOn();
  myObject->setEditOff();
  ```

- `pickable`: if set to true, the object can be manipulated through *draggers* (see section 3.4.6). You can also change this option with the methods

  ```
  myObject->setPickOn();
  myObject->setPickOff();
  ```

Adding 3D objects to the scene

Once a 3D object is loaded or created, it is easy to add it to the viewer's scene:

```
...
PoLAR::Viewer viewer;
...
osg::ref_ptr<PoLAR::Object3D> object = new PoLAR::Object3D(objName);
...
viewer.addObject3D(object.get());
```

You can also remove the object:

```
viewer.removeObject3D(object.get());
```

Or replace an object by another one:

```
viewer.replaceObject3D(oldObject.get(), newObject.get());
```

As like any of OpenSceneGraph objects, you can copy or clone a 3D object:

```
PoLAR::Object3D oldObject;
...
PoLAR::Object3D newObject(oldObject, osg::CopyOp::DEEP_COPY_ALL);
...
PoLAR::Object3D *newObjPtr = oldObject.clone(osg::CopyOp::DEEP_COPY_ALL);
```

Please refer to the `svisu3` and `svisu4` examples for basic usage of loading and adding 3D objects.

3.4.3 Lights

PoLAR provides a class for managing 3D lights. The `PoLAR::Light` class encapsulates the OpenSceneGraph lights. This class provides the methods to easily change the light states (on/off, position, color, etc.).

OpenSceneGraph lights are a very simple encapsulation of OpenGL lights. The `PoLAR::Light` class automatically manages everything related to OpenGL, providing higher-level API than basic OpenSceneGraph lights. More advanced low-level functions are however still available.

There are multiple possibilities to create a light. The easiest way is to call the `PoLAR::Viewer`'s helper method and provide a (X,Y,Z) position:

```
PoLAR::Viewer viewer;
...
float positionX, positionY, positionZ;
...
// the following method creates a light and returns a pointer to it
PoLAR::Light* light = viewer.addLightSource(positionX, positionY, positionZ);
```

You can also create the light first, and only then add it to the scene. You can create one with default values

```
osg::ref_ptr<PoLAR::Light> light = new PoLAR::Light();
viewer.addLightSource(light.get());
```

or pass custom values in the constructor

```
osg::Vec4 position; // the position of the light
osg::Vec4 ambient; // ambient color of the light
osg::Vec4 diffuse; // diffuse color of the light
osg::Vec4 specular; // specular diffuse of the light
osg::ref_ptr<PoLAR::Light> light = new PoLAR::Light(position, ambient, diffuse, specular);
viewer.addLightSource(light.get());
```

or, if you already have a `osg::LightSource`, you can create a `PoLAR::Light` that will use it

```
osg::LightSource* lightSource;
...
osg::ref_ptr<PoLAR::Light> light = new PoLAR::Light(lightSource);
viewer.addLightSource(light.get());
```

You can change the position, color and other options anytime you need to. The `PoLAR::Light` class API offers straightforward methods to do so.
Once the light is created and added to the scene, you can access to it or remove it. If you did not keep a pointer to your light after having created it, you can get it if you know its light number. The light number is an OpenGL feature which associates an integer (between 0 and 7) to a light. This number is unique and ensures a light is unique too. PoLAR automatically associates a light number to a created light. In some cases, it can be useful to provide a number by yourself:

```
osg::ref_ptr<PoLAR::Light> light = new PoLAR::Light();
...
int lightNum = 1;
light->setLightNum(lightNum);
```

Note: If this light number is already used by another light, PoLAR will set an unused light number to your light.

This light number can be used to get a particular light.

```
int lightNum = ....;
PoLAR::Light* light = viewer.getLightSource(lightNum);
// if the light exists
if(light != NULL)
{
 // set the light off
  light->setOff();

 //delete it
  viewer.removeLightSource(light);
```

```
}
else
{
  std::cout << "No such light " << lightNum << std::endl;
}
```

Note 1: By default, OpenSceneGraph creates one light even if the user does not add any lights to the scene. PoLAR keeps this light but links it to the main camera (so that it follows the camera). This default light has the number 0 and can be accessed just like any other lights, and can also be turned off.
Note 2: Standard fixed-function OpenGL cannot handle more than eight lights. That is why PoLAR does not accept more than eight lights.

More about lights

PoLAR lights offer additional features:

- **shadow casting**: using `PoLAR::Light::setShadowCaster(bool b)`, it is possible to make the light cast shadows or not. Note however that, due to OpenSceneGraph limitations, only one light can be a shadow caster in the scene. See section 3.4.5 for more information about shadows.
- **3D viewable representation of the light**: this feature adds a 3D sphere in the scene, representing the light. The 3D sphere is placed at the light's position. It is useful for debugging purposes, or for following the displacement of a light for example. You can add or hide the 3D viewable sphere with `PoLAR::Light::setViewable(bool b)`.
- **Relative or absolute position of the light:** you can choose between setting an absolute of a relative position to the light. The first means the light's position is given in camera coordinates while the latter means its position is given in world coordinates. You can change the reference frame by using these two methods:

```
PoLAR::Light::setReferenceFrameVideo(); // set the position in camera coordinates
PoLAR::Light::setReferenceFrameWorld(); // set the position in world coordinates
```

3.4.4 Other 3D objects

PoLAR provides two other particular objects:

- **Frame axis**: the class `PoLAR::FrameAxis` is a particular 3D object that represents the three axes (X, Y and Z) of the 3D frame. The X axis is represented in red, the Y axis in green, and the z axis in blue.
- **Viewable camera**: Like for the lights, it is possible to add a 3D representation of the main camera. It is useful only in multi-viewer applications, where the main camera of one viewer must be seen as a 3D object in a second viewer. The `PoLAR::ViewableCamera` class provides this 3d representation and makes use of Qt *slots* in order to be easily connectable with actions in a Qt application.

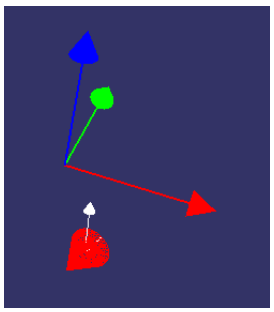

Figure 7: 3D frame axis and viewable camera

These objects are helper objects and are meant to offer the user a fast and easy solution for displaying things that have by default no 3D representation.

3.4.5 Shadows, transparency and phantom objects

This part is dedicated to explain the basics of creating shadowed scenes for augmented reality applications.

Shadows

PoLAR provides the possibility to draw real-time shadows in the 3D scene. On desktop systems, this feature makes usage of the `osgShadow` module, which is part of OpenSceneGraph. This module provides different shadow techniques (e.g. shadow maps, stencil shadows, etc.). PoLAR creates by default a *soft shadow map* technique. It is however possible to set another shadow technique (provided that this shadow technique comes from osgShadow):

```
PoLAR::Viewer viewer;
...
PoLAR::SceneGraph * sceneGraph = viewer.getSceneGraph();
osgShadow::ShadowTechnique* st = /* the new shadow technique  */;
sceneGraph->setShadowTechnique(st);
```

Shadows can be activated through an argument in the `PoLAR::Viewer` constructor.

```
PoLAR::Viewer(QWidget *parent, const char *name, WindowFlags f, bool shadows, bool distortion
    ) ;
```

All these arguments have a default value. Shadows are set to *off* by default.
Shadows can also be turned on and off using methods of the viewer:

```
PoLAR::Viewer viewer;
...
viewer.setShadowsOn();
...
viewer.setShadowsOff();
```

Note: in order to get coherent shadows, it is required to add a custom light source to your 3D scene. The easiest way is by using this method:

```
PoLAR::Viewer viewer;
...
float posX, poY, posZ;
viewer.addLightSource(poxZ, posY, posZ, true);
```

The first three parameters are the X, Y and Z coordinates of the light. The boolean set to `true` means the light will cast shadows.

For Android platforms, since the `osgShadow` module is compatible with the fixed-function pipeline only and thus not available with GLES2, we have developed a simple shadow map generator to replace the missing feature. It only provides standard shadow maps.

Phantom objects

It is possible to create a particular kind of 3D objects called *phantoms*. They are meant to hide the part of the 3D scene behind them, while receiving the shadows of the objects above them.
This kind of objects is very useful in augmented reality because they offer the possibility to mix the background data (data from the real world) with the 3D scene (data from the virtual world).

Let's study one example. First, we want to load a 3D model into a scene and apply a projection matrix to match a background image.

```
int main(int argc,char ** argv)
{
 // Create a viewer with shadows management
 PoLAR::Viewer viewer;
 viewer.setShadowsOn();
 viewer.addLightSource(5,2,5, true);

 // Load background image
 osg::ref_ptr<PoLAR::Image_uc> myImage = new
     PoLAR::Image_uc("image.png", true);
 viewer.setBgImage(myImage);
 viewer.bgImageOn();

// Load a projection matrix
 osg::Matrixd P = PoLAR::Util::
     readProjectionMatrix("project.proj");
 viewer.setProjection(P);

 // Load a 3D model
 PoLAR::Object3D *obj = new PoLAR::Object3D
     ("horse.off", true);
 // optimize it
 obj->optimize();
 // Add the loaded model to the scene
 viewer.addObject3D(obj);

 viewer.show();

 app.connect( &app, SIGNAL(lastWindowClosed()
     ), &app, SLOT(quit()) );
 return app.exec();
}
```

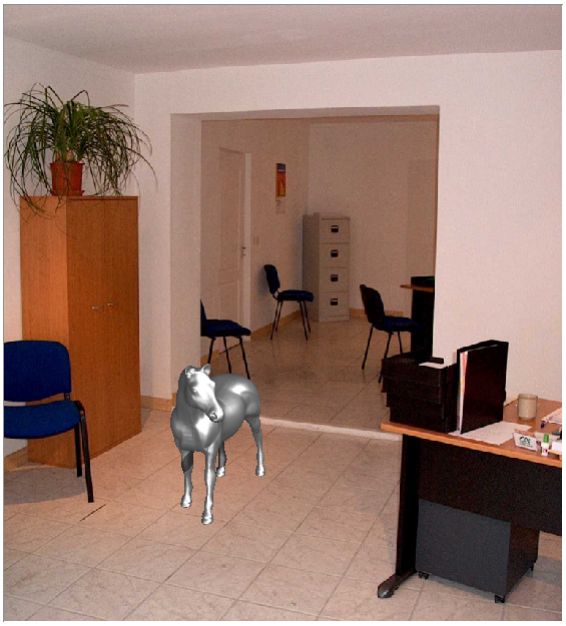

Figure 8: 3D model and projection matrix

The 3D model of the horse was obtained from the Large Geometric Models Archive[16].

We have our 3D model and the projection matrix makes it look nice with the background image. But we have a realism issue: there is no shadow on the ground.
To create this effect, let's add a ground in the 3D scene. PoLAR provides a handy static method that creates a ground.

```
int main(int argc,char ** argv)
{
 ...
 // add a ground
 osg::ref_ptr<PoLAR::Object3D> ground = PoLAR
     ::Object3D::createGround();
 viewer.addObject3D(ground.get());

 viewer.show();
 ...
}
```

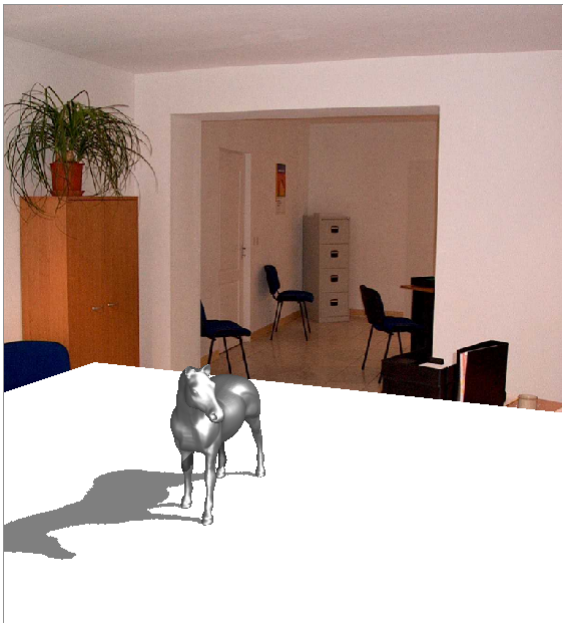

Figure 9: Ground added

[16] http://www.cc.gatech.edu/projects/large_models/

Now we have a ground catching the shadows created by the model. But we occult the background image, which is not what we want.
Our next idea would be to make the ground transparent:

```
int main(int argc,char ** argv)
{
 ...
 // add a ground
 osg::ref_ptr<PoLAR::Object3D> ground = PoLAR
      ::Object3D::createGround();
 viewer.addObject3D(ground.get());

 // make the ground transparent
 ground->setEditOn();
 ground->setTransparencyOn();

 viewer.show();
 ...
}
```

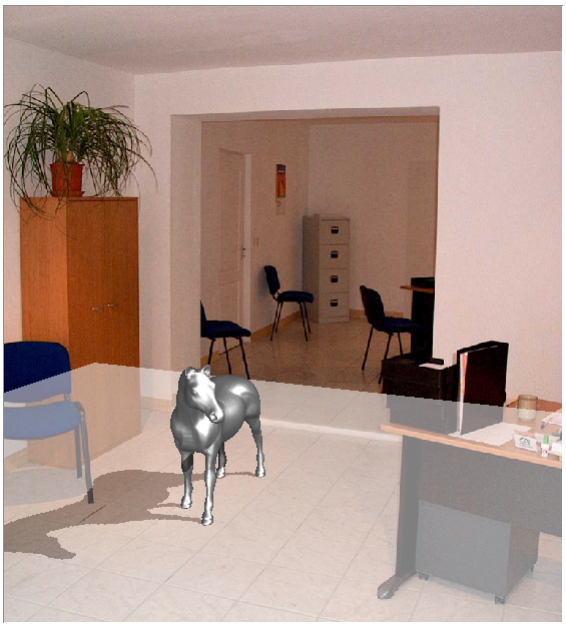

Figure 10: Transparent ground

By default, the method `setTransparencyOn()` sets the transparency to 50%. To make the ground completely disappear, we could set the transparency to 100%:

```
ground->setTransparencyOn();
ground->updateTransparency(0.0);
```

But this would give the same effect as when we had no ground: a completely invisible ground which does not catch the shadows.

The solution is to use *phantom objects*. Setting the ground as phantom is as simple as setting it transparent:

```
int main(int argc,char ** argv)
{
 ...
 // add a ground
 // add a ground
 osg::ref_ptr<PoLAR::Object3D> ground = PoLAR
      ::Object3D::createGround();
 viewer.addObject3D(ground.get());

 // make the ground phantom
 ground->setEditOn();
 ground->setPhantomOn();

 viewer.show();
 ...
}
```

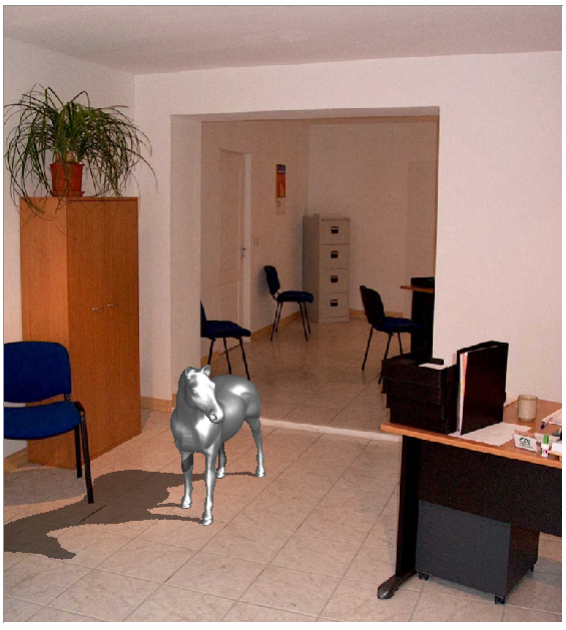

Figure 11: Phantom ground

3.4.6 3D objects manipulation

Like 2D objects, the manipulation of 3D objects is available in one particular mode: the `OBJECT3D` mode. You can refer to the more advanced examples `testpick`, `testvisucolor3D` and `testvisucolor3D2`.

Camera manipulation
We call *camera manipulation* the possibility to move the main camera in the scene. This feature uses a *camera manipulator* to be able to manipulate the camera. PoLAR provides a custom camera manipulator:

```
...
PoLAR::SceneGraphManipulator *m = new PoLAR::SceneGraphManipulator;
viewer.setCameraManipulator(m);
...
```

But all OpenSceneGraph native manipulators can be used. Please refer to the OpenSceneGraph documentation, in particular to the `osgGA` namespace[17].
Note: if the user does not add his own manipulator, PoLAR creates a `PoLAR::SceneGraphManipulator` when the first 3D object is added to the scene. It is thus not necessary to explicitely add a camera manipulator in your application.

Now, to activate the `OBJECT3D` manipulation mode, use this method:

```
viewer.startManipulateSceneSlot();
```

Then, if you use the custom `PoLAR::SceneGraphManipulator`, different manipulations are available. Default commmands are:

- **mouse left button**: rotate around the center
- **mouse middle button**: do a 3D translation parallely to the image plane (XY)
- **mouse right button**: do a 3D translation orthogonaly to the image plane (Z)
- **spacebar**: reset the view to the initial extrinsics

These are the default commands. They can be modified with these methods which have to be set directly to the manipulator:

```
PoLAR::SceneGraphManipulator::setRotateCameraMouseButton(osgGA::GUIEventAdapter::
    MouseButtonMask button);
PoLAR::SceneGraphManipulator::setTranslateXYMouseButton(osgGA::GUIEventAdapter::
    MouseButtonMask button);
PoLAR::SceneGraphManipulator::setTranslateZMouseButton(osgGA::GUIEventAdapter::
    MouseButtonMask button);
PoLAR::SceneGraphManipulator::setResetCameraShortcut(osgGA::GUIEventAdapter::KeySymbol key);
```

In addition to the camera manipulation, it is possible to interact directly with one 3D object in the scene. PoLAR provides two types of handlers: *object edition* and *geometry edition* manipulators.

Object manipulation

It is possible to manipulate 3D objects with the mouse in a WYSIWYG way. This is made possible thanks to the OpenSceneGraph *draggers*. For more information about these draggers, please visit refer to the `osgManipulator::Dragger` class in OpenSceneGraph.
There are multiple draggers available: `osgManipulator::TabBoxDragger, TrackballDragger, TranslateAxisDragger, ScaleAxisDragger, RotateSphereDragger`. Once one of them is added to an object, a geometrical shape appears. You can use this shape to manipulate the object: scale around the three axes, rotation, position, depending on which pick handler is used.

- The `TabBox` allows you to move and/or scale the object along the three directions, by manipulating the faces of the box, its edges, or its vertices.
- The `TrackBall` allows you to rotate the object around the three axes (one axis at a time if you click on the circles, or in a free way if you click between the circles).
- The `TranslateAxis` allows you to move the object along the three directions by pulling the axes.

[17] `http://trac.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/a01597.html`

- The `ScaleAxis` allows you to scale the object along the three directions by pulling the axes.
- The `RotateSphere` allows you to rotate the object in all directions by manipulating the sphere.

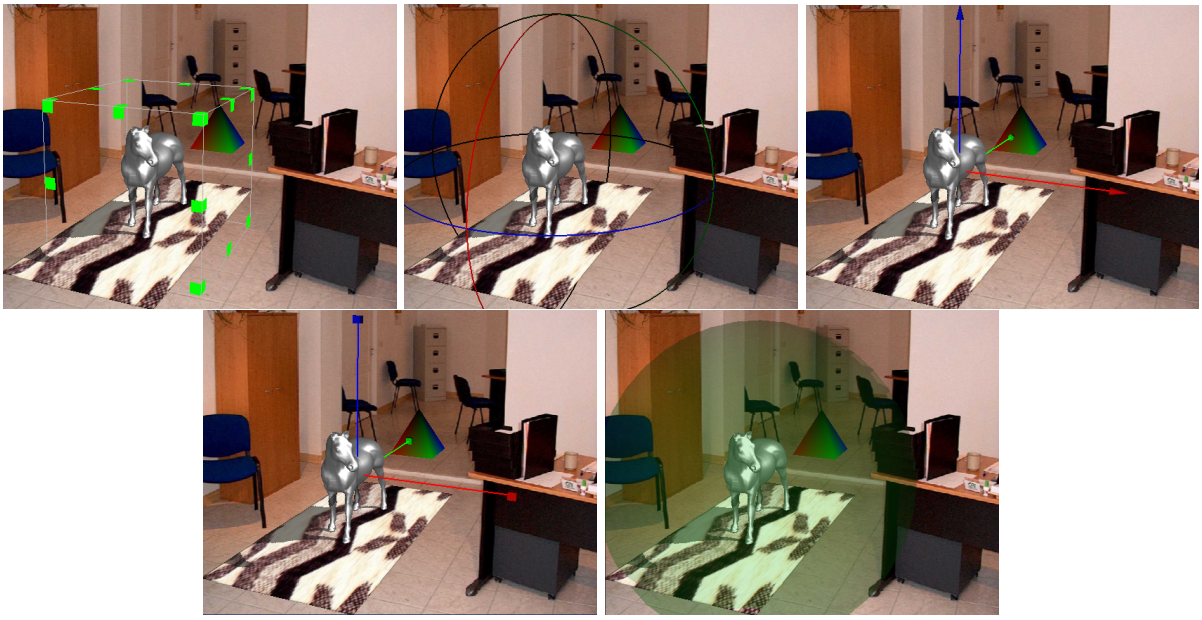

Figure 12: The different 3D object draggers
top left to bottom right: TabBox, TrackBall, TranslateAxis, ScaleAxis, RotateSphere

The following lines show how to add a `TabBox` dragger to an object:

```
PoLAR::Object3D* object;
...
osg::ref_ptr<osgManipulator::Dragger> dragger;
dragger = new osgManipulator::TabBoxDragger;
object->addDragger(dragger.get());
```

You can remove a dragger with this line:

```
object->removeDragger();
```

Object picking

Object picking is the possibility to pick an object in the scene with the mouse in order to select it and manipulate it. The difference with the previous paragraph is that there is no need to manually add a dragger to an object: the action of picking an object adds the dragger.
The object edition pick handler is a reimplementation of the OpenSceneGraph `GUIEventHandler`. The base class is abstract, only the inherited classes can be instantiated.

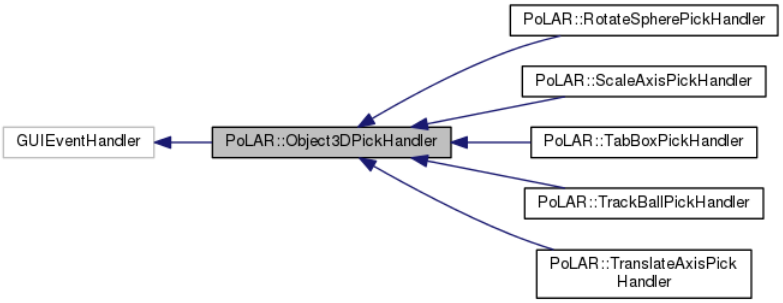

Figure 13: Object3DPickHandler inheritance diagram

These types are actually reimplemented from the OpenSceneGraph manipulators of the same names.

To activate object picking you just have to add the related handler to the viewer:

```
PoLAR::Object3DPickHandler *handler;
// the type of handler created can depend on something else
if(...)
   handler = new PoLAR::TabBoxPickHandler;
else
   handler = new PoLAR::TranslateAxisPickHandler;
viewer.addEventHandler(handler);
```

When in `OBJECT3D` interaction mode, the default commands to use the object pick handler are:

- **Numpad '1'** to activate it
- **Numpad '0'** to go back in camera manipulation mode

These commands are customizable through these methods:

```
PoLAR::Object3DPickHandler::setObjectEditionModeKey(osgGA::GUIEventAdapter::KeySymbol key);

PoLAR::Object3DPickHandler::setCameraManipulationModeKey(osgGA::GUIEventAdapter::KeySymbol
    key);
```

Geometry edition

The **geometry pick handler** is made for manipulating the vertices of the object.

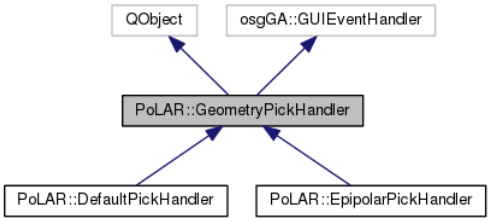

Figure 14: Object3DPickHandler inheritance diagram

Like for the `PoLAR::Object3DPickHandler`, the base class is abstract, and only the inherited classes are instantiable. The `DefaultPickHandler` provides vertex manipulation in the three axes X, Y and Z (following the camera frame). the `EpipolarPickHandler` provides vertex manipulation in the three axes X, Y (following the camera frame) and Z (following a line going from this vertex to the projection of a chosen camera focal point).

Now, you just have to add the handler to the viewer:

```
PoLAR::DefaultPickHandler *handler = new PoLAR::DefaultPickHandler;
viewer.addEventHandler(handler);
```

When in `OBJECT3D` interaction mode, the default commands to use the geometry pick handler are:

- **Numpad '2'** to activate it
- **Numpad '0'** to go back in camera manipulation mode
- **W** to save to currently picked vertex information
- **hold CTRL** to activate the vertex manipulation mode. Release it to stop the manipulation
- When CTRL is pressed, the **mouse left button** translates the vertex in the X and Y directions
- When CTRL is pressed, the **mouse middle button** translates the vertex in the Z direction.

These commands are of course also customizable using these methods:

```
PoLAR::GeometryPickHandler::setGeometryEditionModeKey(osgGA::GUIEventAdapter::KeySymbol key);

PoLAR::GeometryPickHandler::setCameraManipulationModeKey(osgGA::GUIEventAdapter::KeySymbol
    key);

PoLAR::GeometryPickHandler::setSaveManipulatedVertexInfoKey(osgGA::GUIEventAdapter::KeySymbol
     key);

PoLAR::GeometryPickHandler::setManipulateVertexModifier(osgGA::GUIEventAdapter::KeySymbol
    modifier);

PoLAR::GeometryPickHandler::setUsingTranslate1DVertexDraggerMouseButton(osgGA::
    GUIEventAdapter::MouseButtonMask button);

PoLAR::GeometryPickHandler::setUsingTranslate2DVertexDraggerMouseButton(osgGA::
    GUIEventAdapter::MouseButtonMask button);
```

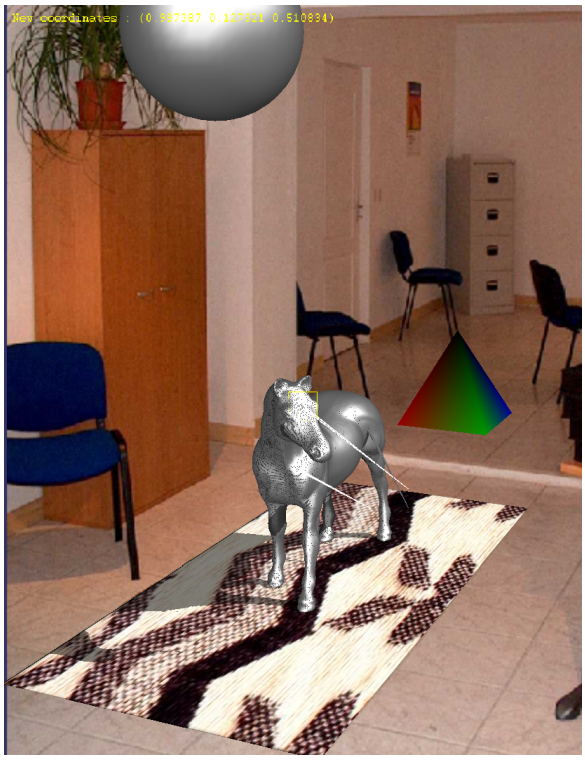

Figure 15: Example of geometry edition: vertex displacement

3.4.7 Projection matrices

Camera projection matrices are matrices applied to the 3D camera in order to change its intrinsics and extrincics parameters (i.e. to change the way how the 3D scene is rendered on screen). Projection matrices are used in augmented reality to make a 3D scene match a real one by applying on the virtual camera the same parameters as the real one has. They are also used in medical imaging in order to adapt the virtual scene to a particular medical device, e.g. to model the behaviour of X-ray generators.

Basic use case

In most simple applications, a default matrix is enough. The `PoLAR::Viewer` can create a matrix adapted to your 3D scene. If you want to set the camera automatically centered on your 3D scene so that your whole scene is displayed on screen, call this method after the creation of your 3D scene:

```
...
viewer.setProjection();
...
```

If you prefer to set a particular width and height for delimiting a frame in which the whole 3D scene is displayed, you can call this method:

```
...
// PoLAR uses the OpenSceneGraph osg::Matrix type
osg::Matrixd P = viewer.createVisionProjection(width, height);
viewer.setProjection(P);
...
```

Custom matrices

Computation of projection matrices **is not** part of PoLAR. However PoLAR is able to load and save projection matrices created from another source. You can provide a file in which a projection matrix is saved:

```
...
osg::Matrixd P = PoLAR::Util::readProjectionMatrix("projection.txt");
viewer.setProjection(P);
...
```

In the saved file, the projection matrix should look like this example:

```
587.41397786473419273  520.25534080362365330  -33.24033400989228681   250.04555996022725139
-58.06394433515148278  183.05966990630986401  -772.69234269060245878  1831.32110646900514439
-0.34957611629405105   0.92393233588785639    -0.15538847324553320    3.52177131159614820
```

It is a 3x4 matrix. Nothing else must be present in the file.

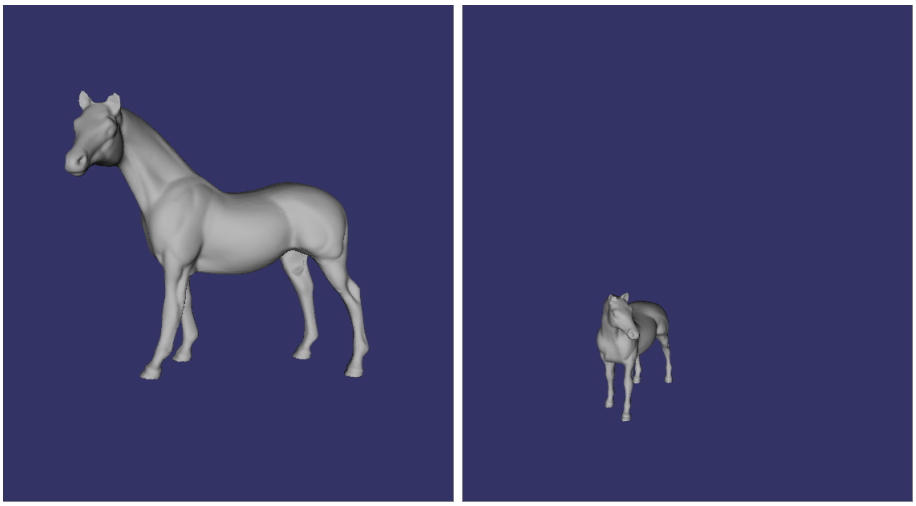

Figure 16: The same scene, with the default projection matrix *(left)* and with a custom projection matrix *(right)*

PoLAR can also save the current projection matrix of the scene camera. This is useful for example when you want to save the matrix after you have manipulated the scene camera *(see section 3.4.6)*.

```
...
// get the projection matrix
osg::Matrixd P = viewer.getProjection();
//now save it
PoLAR::Util::saveProjectionMatrix(P, "projection.txt");
// you can also simply print it out (you will need to include the <osg/io_utils> header)
std::cout << P;
...
```

You can then use the saved matrix in other programs.

Types of projection

There is also a possibility to *choose a type of projection*. Note however that, except in some very particular cases, **you will not need to change the default type**.

The different types are: `VISION` (default), `ANGIO`, and `ANGIOHINV`. The `ANGIO` type represents an angiographic projection (medical radiography). `ANGIOHINV` represents an angiographic projection with horizontal inversion. These two types are specific to medical imaging and should not be used in most cases.
To set a particular type, you just need to set it as second parameter of the `setProjection()` method:

```
...
osg::Matrixd P = PoLAR::Util::readProjectionMatrix("projection.txt");
PoLAR::Viewer::ProjectionType pt = PoLAR::Viewer::VISION;
if(...) // particular medical imaging case
   pt = PoLAR::Viewer::ANGIO;
viewer.setProjection(P, pt);
...
```

Note that the methods used for the creation of basic matrices as shown at the beginning of this section actually use the `VISION` type.

3.4.8 Animated 3D objects

PoLAR takes advantage of the OpenSceneGraph `osgAnimation`[18] module to manage object animations. This is particularly useful when you have an animated object in an external software such as Blender[19] and you want to add it to your 3D scene.
In this section we explain how to export an animated object from Blender, and how to use it in your PoLAR application. We will not explain how to *create* an animation in Blender, since this goes out of the scope of the PoLAR manual.

Exporting from Blender

First you need to install the correct Blender exporter. You can find it here:
`https://github.com/cedricpinson/osgexport`
It comes as a zip file. In Blender, go to the User Preferences, and in the `Add-ons` tab click on `Install from File`. Select the zip file you have downloaded and everything should go fine.

In order for the exporter to make the correspondence between the armature bones and the vertices, each vertex group must have the exact same name than the bone it is attached to. Vertex groups are visible in the `Properties / Object Data / Vertex Groups` tab.

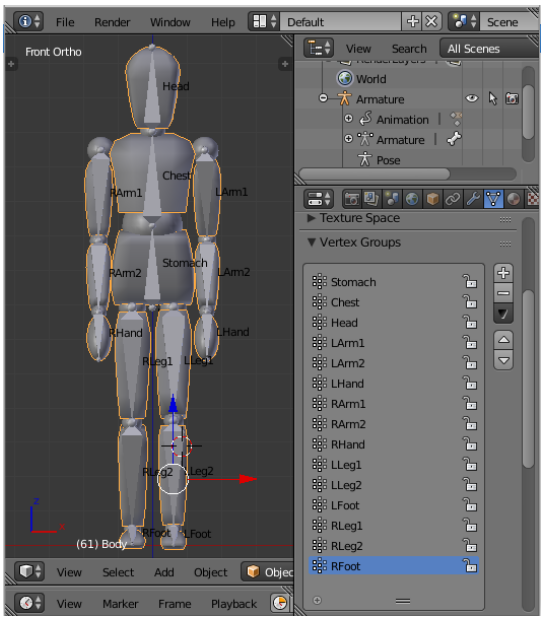

Figure 17: Correspondence between bones names and vertex groups names

[18] `http://trac.openscenegraph.org/projects/osg/wiki/Community/NodeKits/osgAnimation`. This page may be out of date.
[19] `http://www.blender.org`

Now, assuming your animation is made (with the keyframes correctly set), you just need to export the object as `OSG Model`. The exported object extension is `.osgt`.

Note: if you plan to use your model on mobile platform, it is advised to generate triangular faces instead of quads. You can triangulate your faces in Blender by selecting all your mesh in edit mode, and click on `Mesh->Faces->Triangulate faces`.

Loading in PoLAR

Loading an animated object in PoLAR is as straightforward as loading a classic 3D object. The only difference is the class name:

```
#include <PoLAR/AnimatedObject3D.h>
...
PoLAR::Viewer viewer;
osg::ref_ptr<PoLAR::AnimatedObject3D> object = new PoLAR::AnimatedObject3D("model.osgt");
viewer.addObject3D(object.get());
```

All the `PoLAR::Object3D` constructor options are also available for the `PoLAR::AnimatedObject3D` class.

Playing the animation

Once the object is loaded, the animation can be started:

```
// change the speed of the animation
object->setAnimationSpeed(1.5);
// start playing the animation
object->play();
...
// stop the animation
object->stop();
```

By default the animations are set to be played in loop mode. You can change the play mode:

```
// change play mode to single shot
object->setAnimationPlayMode(osgAnimation::Animation::ONCE);
```

Multiple animations

More advanced users may want to play multiple animations on one model. Provided that the exported model has these animations, it is possible to access the different animations:

```
// access to and play next animation
object->selectNextAnimation();
object->play();
...
// access to and stop previous animation
object->selectPreviousAnimation();
object->stop();

// start playing the animation with the given name
object->playByName("myAnimation");

// these lines have the same effect than the previous one
bool b = object->selectByName("myAnimation");
if(b) object->play();
```

Note that most methods return booleans which can be used to check if the method has successfully run, or if an error (e.g. missing animation) has been found.
See the `animation` example in the Examples folder for a more detailed example of using an animated object.

4 Binding a tracking algorithm

4.1 Overview

In augmented reality, a tracking algorithm is a way to compute a projection matrix from an image, so that the 3D scene matches the real scene.
One of the main features of PoLAR is that it is completely independent from tracking algorithms, leading in the possibility to use any algorithm you would like. The API is made to be the simple and intuitive. This section aims at explaining the basics, but also most advanced usage of binding a tracking algorithm to PoLAR.

This feature takes advantage of the powerful Qt *signal and slot* system. For more information, please read the official documentation[20].

4.2 Basic application

This example shows a basic PoLAR application with the simplest way to create a viewer, a camera manager, how to add a simple 3D object, a phantom ground, and the simplest way to connect a tracking algorithm:

```
#include <QApplication>
#include <PoLAR/Viewer.h>
#include <PoLAR/Image.h>
#include <PoLAR/VideoPlayer.h>
#include <PoLAR/Object3D.h>

int main(int argc, char** argv)
{
    // create the Qt application
    QApplication app(argc, argv);

    //create the PoLAR viewer
    PoLAR::Viewer viewer;

    // create the camera manager
    osg::ref_ptr<PoLAR::VideoPlayer> videoStream = new PoLAR::VideoPlayer(0);
    osg::ref_ptr<PoLAR::Image_uc> myImage = new PoLAR::Image_uc(videoStream);
    viewer.setBgImage(myImage);
    viewer.bgImageOn();
    videoStream->play();

    // add a 3D object to the scene
    osg::ref_ptr<PoLAR::Object3D> obj = new PoLAR::Object3D("data/chair.obj");
    viewer.addObject3D(obj);

    // add a phantom ground
    osg::ref_ptr<PoLAR::Object3D> ground = PoLAR::Object3D::createGround();
    ground->setPhantomOn();
    viewer.addObject3D(ground);

    // connect the video stream to your tracking algorithm
    QObject::connect(videoStream.get(), &PoLAR::VideoPlayer::newFrame,
                     [=](unsigned char* data, int w, int h, int d)
    {
        /* Your tracking code here */
        osg::Matrix M = /* The computed projection matrix */;
        viewer.setProjection(M);
    });

    // show the viewer
    viewer.show();

    // run the Qt application
    app.connect(&app, SIGNAL(lastWindowClosed()), &app, SLOT(quit()));
    return app.exec();
}
```

[20] http://doc.qt.io/qt-5/signalsandslots.html

Let's focus on the important part:

```
QObject::connect(videoStream.get(), &PoLAR::VideoPlayer::newFrame,
                 [=](unsigned char* data, int w, int h, int d)
{
    /* Your tracking code here */
    osg::Matrix M = /* The computed projection matrix */;
    viewer.setProjection(M);
});
```

This code connects the signal `PoLAR::VideoPlayer::newFrame` to a *lambda function* slot in which you just have to replace the comments by your tracking code.
The signal has 4 parameters:

- `unsigned char * data` is a pointer to the array containing the pixels of the current frame.
- `int w` is the width of the frame.
- `int h` is the height of the frame.
- `int d` is the depth of the frame: possible values are 1 (grayscale image), 3 (RGB image), or 4 (RGBA image) but in a normal usage it should always be 3.

Thanks to these parameters you can convert the incoming frame to the format needed for your tracking algorithm.

Your algorithm should return a projection matrix, in the same format as explained in section 3.4.7, which corresponds to a typical projection matrix notation in computer vision. Note however that it must be converted into a `osg::Matrix` for compatibility purpose.
You can set the full projection matrix with the method `viewer.setProjection`, but you can also set both extrinsics and intrinsics matrices separately:

```
osg::Matrix I;
osg::Matrix E;
...
// set intrinsics matrix
viewer.setIntrinsics(I);
// set pose (extrinsics matrix)
viewer.setPose(E);
```

4.3 Connect the video manager with another class

In bigger applications it can be necessary to define a class that will manage the tracking. You will need a method in this class which will receive the frame update, instead of using a lambda function as explained in the previous section. This will require to declare the class following some special notation.
First, create the class you would like to use to receive the frame updates:

```
#include <QObject>

class MyTrackingClass : public QObject
{
  Q_OBJECT

  public:
  ...

  public slots:

  void newFrameReceived(unsigned char* data, int w, int h, int d)
  {
    /* your tracking algorithm here */
  }

};
```

Your class should inherit from `QObject`, which is the base class of Qt objects. The `Q_OBJECT` macro enables the use of signals and slots in your class. The `newFrameReceived` method is declared as a `public slot`, which means it can receive signals from outside the class. It must respect the same method parameters as the `PoLAR::VideoPlayer::newFrame` signal. At the end of this method, you will have to manage the projection matrix update of the PoLAR::Viewer. One possibility is to set the viewer as a class argument:

```
class MyTrackingClass : public QObject
{
  Q_OBJECT

  public:
  // constructor
  MyTrackingClass(PoLAR::Viewer* viewer):
    _viewer(viewer)
  {}

  public slots:
  void newFrameReceived(unsigned char* data, int w, int h, int d)
  {
    /* your tracking algorithm here */
    osg::Matrix M = /* the computed projection matrix */;
    _viewer->setProjection(M);
  }

  private:
  PoLAR::Viewer* _viewer;

};
```

Now in the main function of your application, you can create a video manager and an object of your custom class, and connect them together:

```
// viewer creation
PoLAR::Viewer viewer;

// video manager creation
osg::ref_ptr<PoLAR::VideoPlayer> videoStream = new PoLAR::VideoPlayer(0);
osg::ref_ptr<PoLAR::Image_uc> myImage = new PoLAR::Image_uc(videoStream);
viewer.setBgImage(myImage);
viewer.bgImageOn();
videoStream->play();

// custom tracking class creation
MyTrackingClass* myClass = new MyTrackingClass(&viewer);

// connecting the video manager to the tracker class
QObject::connect(videoStream, SIGNAL(newFrame(unsigned char*,int,int,int)),
         myClass, SLOT(newFrameReceived(unsigned char*,int,int,int)));
```

Such software conception is very convenient, for example to switch easily between different tracking classes (using different algorithms). Indeed, you just have to change the object on which the `PoLAR::VideoPlayer::newFrame` signal is connected to change the tracking algorithm in use.

5 Physics

This section covers a major feature of PoLAR: the possibility of binding PoLAR with physics engines.

Simulating physical interactions between objects is often needed in augmented reality to add some realism in the scenes. On the other hand, medical imaging applications request physically correct deformations of soft bodies. Various physics engines exist but they generally consist in a trade off between fast and accurate solutions.
Starting from this observation, we have decided to offer both capacities by enabling the possiblity to choose and switch between different physics engines. As of today, two different engines are bound with PoLAR:

- the **Bullet**[21] physics engine which offers a fast solution of rigid bodies interaction and a physically simplified model of soft bodies based on mass-spring systems;
- the **VegaFEM**[22] physics library which offers large deformations modeling through a finite element method.

We have developped our bindings with the idea to keep a sufficient level of abstraction from the physics engine. This gives the possibility to use the same programming technics regardless of the engine used.
It is however important to use the most adapted physics engine. Bullet is adapted to AR scenes that need rigid body or simplified soft body interactions with collisions, without looking for a physically realistic behaviour. VegaFEM is adapted for the visualisation of large soft bodies such as organs. It does not provide collisions. VegaFEM is more adapted to solve physical systems without seeking for a real-time computing.

5.1 Installation

The use of physics engines is totally optional, and you can choose to use either one or both proposed engines. Note that even if PoLAR provides the basic tools, it is necessary to know at least a little of how the physics engine works in order to set up a working scene.

5.1.1 Using Bullet

As of today, Bullet has been tested on Linux and Mac platforms only. On these platforms, it is available in the official package repositories. For example on Ubuntu 16.04, the related packages are `libbullet-dev` and its additional packages that will automatically be installed. On Windows and mobile platforms the use of Bullet is possible with a compilation from the sources but this has not been tested along with PoLAR.
Once installed, re-run CMake in the PoLAR project with the CMake variable `USE_BULLET` set to `ON` and compile.

5.1.2 Using VegaFEM

The only way to use VegaFEM is to compile it from the sources. You can download them from the official website: `http://run.usc.edu/vega/download.html` and follow the instructions given with the sources.
Once the compilation is completed, you will need to add an environment variable to your path so that PoLAR can find VegaFEM: set the variable `VegaFEM_BASE_DIR` to the root directory of your installation of VegaFEM.
Now you need to recompile PoLAR with the VegaFEM module. Set the PoLAR CMake variable `USE_VEGA` to `ON` and compile.

[21] `http://bulletphysics.org/`
[22] `http://run.usc.edu/vega/`

5.2 Generalities

5.2.1 Simulation thread

Physics computation is made in a separate threaded loop. This ensures the physics computation does not interfere with the GUI and keep a responsive user interface even when the physics require heavy computation.

There are two specific classes needed to set up a simulation loop:

- the *simulation manager* which is responsible for launching and stopping the simulation thread, and manages the displayable information on screen (e.g. frequency of the loop)
- the *simulation worker* which runs the threaded loop.

Users should interact with the simulation manager and not directly with the worker. The manager is an interface to the workers and is the same regardless of the physics engine. The worker is however dependent of the engine since the jobs accomplished in the loop interact with the engine and can change with the application.

5.2.2 Physical 3D objects

Alongside with the graphical representation of the objects that we already know (see section 3.4), it is needed to generate a physical representation of the objects in order to have them interact between each other.
In the case of Bullet, the physical version of the object can be either manually created and then linked to the graphical object, or either procedurally generated from the graphical object. In the case of Bullet soft bodies, a third possibility is to generate a graphical object procedurally from the physical Bullet object (useful when the physical object is created from Bullet helper methods).
VegaFEM needs tetrahedral or hexahedral meshes for the physical computations. That is why VegaFEM objects come in two files, one for the graphical representation and one for the physical representation.

5.3 Create your first Bullet scene

Bullet provides powerful rigid body interactions and collisions, and also simplistic deformable soft bodies.
PoLAR provides the basic tools needed to create a Bullet scene (including the set up of default environment settings).
Please refer to the example `demoBullet` for the related working example.

5.3.1 Setting up the scene parameters

First, we need to create the Bullet scene and set parameters to it. The following code sets up and returns an object of the class `btSoftRigidDynamicsWorld`. This class represents the physical world and includes default constraint solver and collision configuration.

```
#include <PoLAR/BulletUtil.h>
...
// Bullet world
btSoftRigidDynamicsWorld *bw = PoLAR::Bullet::initBulletWorld();
```

This basic setup is adapted for soft and rigid body dynamics.
We will later link our physical objects to the `btSoftRigidDynamicsWorld` object.

Note: world parameters like gravity or air density are set to real life values by default (gravity to -10 along Z, air density to 1.2, etc.). It is possible to change them by accessing the `btSoftBodyWorldInfo` object:

```
btSoftBodyWorldInfo* worldinfo = PoLAR::Bullet::DynamicsWorld::getWorldInfo();
// apply the air density
worldinfo->air_density = btScalar(value);
```

```
btVector3 gravity(xVal, yVal, zVal);
// apply gravity on the btSoftBodyWorldInfo and on the btSoftRigidDynamicsWorld
worldinfo->m_gravity = gravity;
bw->setGravity(gravity);
```

5.3.2 Creating and adding physics objects

There are three different ways of creating a 3D object with physics features:

- 1. Create a visual 3D model and generate a physics object from it. This works for rigid and soft bodies.
  First let's create a rigid body from a cube:

```
PoLAR::Viewer viewer;
btSoftRigidDynamicsWorld *bw = PoLAR::Bullet::initBulletWorld();
...
// Load a 3D object from a file and set a mass of 10 to it
float mass = 10.0f;
osg::ref_ptr<PoLAR::BulletObject3D> cube = new PoLAR::BulletObject3D("cube.obj", mass);
// Generate the rigid body
cube->createRigidBody();
// Add the object to the PoLAR::Viewer
viewer.addObject3D(cube);
// Add the object to the Bullet world
bw->addRigidBody(cube->getRigidBody());
```

  And now let's create a soft body from a sphere.

```
// Load the 3D object from a file. The default mass is 1
osg::ref_ptr<PoLAR::BulletSoftObject3D> sphere = new PoLAR::BulletSoftObject3D("sphere.
    obj");
// Generate the soft body
sphere->createSoftBody();
// The following lines are pure Bullet stuff: they change the object's linear stiffness
btSoftBody *softbody = sphere->getBtSoftBody();
btSoftBody::Material* pm = softbody->appendMaterial();
pm->m_kLST = 0.45;
// Generate bending constraints between the vertices
softbody->generateBendingConstraints(5, pm);
// Add the object to the PoLAR::Viewer
viewer.addObject3D(sphere);
// Add the object to the Bullet world
bw->addSoftBody(softbody);
```

- 2. Create a Bullet physics object and generate a visual object from it. This option only works with soft bodies.
  In this example we create a flag that moves with the wind:

```
// Get access to the world info parameters
btSoftBodyWorldInfo& worldInfo = *(PoLAR::Bullet::DynamicsWorld::getWorldInfo());

// Create the soft body using a Bullet helper function.
const short resX(12), resY(9);
const osg::Vec3 llCorner(-2., 0., 5.);
const osg::Vec3 uVec(4., 0., 0.);
const osg::Vec3 vVec(0., 0.1, 3.);
btSoftBody* softBody( btSoftBodyHelpers::CreatePatch(worldInfo,
                                                     PoLAR::Bullet::asBtVector3(
                                                         llCorner),
                                                     PoLAR::Bullet::asBtVector3(
                                                         llCorner + uVec),
                                                     PoLAR::Bullet::asBtVector3(
                                                         llCorner + vVec),
                                                     PoLAR::Bullet::asBtVector3(
                                                         llCorner + uVec + vVec),
                                                     resX, resY, 1+4, true));

// Configure the soft body for interaction with the wind.
softBody->getCollisionShape()->setMargin(0.1);
softBody->m_materials[0]->m_kLST = 0.3;
softBody->generateBendingConstraints(2, softBody->m_materials[0]);
```

```
softBody->m_cfg.kLF = 0.05;
softBody->m_cfg.kDG = 0.01;
softBody->m_cfg.piterations = 2;
softBody->m_cfg.aeromodel = btSoftBody::eAeroModel::V_TwoSidedLiftDrag;
softBody->setWindVelocity(btVector3(50.0, 0.0, 0.0));
softBody->setTotalMass(1.0);

// Create a PoLAR object from this soft body
osg::ref_ptr<PoLAR::BulletSoftObject3D> flag = new PoLAR::BulletSoftObject3D(softBody);
// Add the object to the PoLAR::Viewer
viewer.addObject3D(flag);
// Add the object to the Bullet world
bw->addSoftBody(softBody);
```

- 3. Bind an already existing physics object to an already existing visual object.

```
btSoftBody* softBody;
osg::Node* node;
...
// Create the object from the OSG node and the Bullet object
osg::ref_ptr<PoLAR::BulletSoftObject3D> object = new PoLAR::BulletSoftObject3D(node,
    softBody);
// Add the object to the PoLAR::Viewer
viewer.addObject3D(object);
// Add the object to the Bullet world
bw->addSoftBody(softBody);
```

Option #1 is the most common way and works nicely for rigid bodies and and very basic soft bodies with simple geometries like spheres. Option #2 is suitable when the visual representation is easier to generate than the physical part, in particular when the physical object is generated thanks to a Bullet static method. Option #3 is generally not necessary and is used at your own risks. The most important is to be sure that the physical object and the visual object have the same number of vertices.

5.3.3 Creating the simulation manager

Once the scene is set up, it is time to create the *simulation manager* which will launch and manage the simulation.

```
#include <PoLAR/SimulationManager.h>
#include <PoLAR/BulletWorker.h>
...

// Get a pointer to the Bulelt world
btSoftRigidDynamicsWorld* bw = PoLAR::Bullet::DynamicsWorld::getDynamicsWorld();

// Create the worker, i.e. the class that performs the simulation loops
osg::ref_ptr<PoLAR::BulletWorker> worker = new PoLAR::BulletWorker(bw);

//Create the simulation manager
osg::ref_ptr<PoLAR::SimulationManager> manager = new PoLAR::SimulationManager(worker);
// Launch the thread
manager->launch();
```

With these lines, the simulation will start as soon as you run your program, at a framerate of 30 Hz. You can change this behaviour with the following commands:

```
// set the worker not to launch automatically
worker->setLaunchOnStart(false);

// Set the time step to 20.0 ms (so a framerate of 50 Hz)
worker->setTimeStep(20.0);
```

You can control the simulation thanks to the manager's API:

```
// run the simulation
manager->play();

//pause the simulation
manager->pause;
```

5.4 Create your first VegaFEM simulation

VegaFEM offers powerful soft body deformations based on finite element methods. On the same model as for Bullet, PoLAR provides some tools to set up a VegaFEM-based simulation.
Please refer to the example `demoVega` for the related working example.

5.4.1 Creating a VegaFEM-ready 3D object

VegaFEM needs two different object representations in order to work:

- the physical representation, generally provided as a `.veg` file, and which is made of tetrahedral or hexahedral elements
- the graphical representation, for example a simple `.obj` file, whose number of vertices must be the same as the number of vertices in the physical representation.

You can create a PoLAR 3D object for VegaFEM thanks to these two files.

```
#include <PoLAR/VegaObject3D>

PoLAR::Viewer viewer;
...
osg::ref_ptr<PoLAR::VegaObject3D> object = new PoLAR::VegaObject3D("mesh.obj", "mesh.veg");
viewer.addObject3D(object.get());
```

Now that the object is created, we can set physics parameters to the simulation.

5.4.2 Physics parameters

VegaFEM physics is a bit more difficult to set up than a Bullet world. We advise refering to the official VegaFEM user's manual[23] first.
The major VegaFEM class is the `IntegratorBase` class, which is a generic class for managing the simulation integrator.
Here we create a `ImplicitNewmarkSparse` integrator (which is derived from `IntegratorBase`) from a tetrahedral mesh:

```
TetMesh *tetraMesh = (TetMesh*)object->getVolumetricMesh();
CorotationalLinearFEM * deformableModel = new CorotationalLinearFEM(tetraMesh);

//Apply force, generate matrices
ForceModel * forceModel = new CorotationalLinearFEMForceModel(deformableModel);
int numTetraVertices = 3 * tetraMesh->getNumVertices(); // total number of DOFs

SparseMatrix *massMatrix;
GenerateMassMatrix::computeMassMatrix(tetraMesh, &massMatrix, true);
int positiveDefiniteSolver = 0;
int numConstrainedDOFs = 0;
int *constrainedDOFs = NULL;
double dampingMassCoef = 1.0;
double dampingStiffnessCoef = 0.01;
double timeStep = 0.2; // the timestep, in seconds
float newmarkBeta = 0.25;
float newmarkGamma = 0.5;
int maxIterations = 1;
double epsilon = 1E-6;
int numSolverThreads = 1;
ImplicitNewmarkSparse *integrator = new ImplicitNewmarkSparse(numTetraVertices, timeStep,
    massMatrix, forceModel, positiveDefiniteSolver, numConstrainedDOFs, constrainedDOFs,
    dampingMassCoef, dampingStiffnessCoef, maxIterations, epsilon, newmarkBeta, newmarkGamma,
     numSolverThreads);
```

For more information about what each parameter corresponds to, please refer to the VegaFEM docmumentation.
Finally apply the integrator to the Vega object:

```
object->setIntegrator(integrator);
```

[23] `http://run.usc.edu/vega/download.html`

5.4.3 Creating the simulation manager

Just like Bullet simulation managers, you need to create a simulation managerto run a VegaFEM simulation with PoLAR. Since every simulation can be different (unlike a Bullet simulation), it can be necessary to inherit from the provided pattern and adapt it to the current usage:

```
#include <PoLAR/SimulationManager.h
#include <PoLAR/VegaWorker.h

class MyVegaWorker: public PoLAR::VegaWorker
{
public:
   MyVegaWorker(PoLAR::VegaObject3D* object, IntegratorBase* integrator):
       VegaWorker(integrator), _object(object) {}

protected:
   void initialize()
   {
       _object->initForces();
   }

   void run()
   {
       // get the accumulation of forces applied to the object
       double* forces = _object->applyForces();

       // apply the forces set to the integrator
       setExternalForcesToIntegrator(forces);

       // perform one timestep
       int code = doTimestep();
       if (code != 0)
       {
           std::cerr << "The integrator went unstable. Reduce the timestep, or increase the
               number of substeps per timestep." << std::endl;
       }

       // request update of the visual object
       _object->requestMeshUpdate();
   }

   // local pointer to the simulated object
   osg::ref_ptr<PoLAR::VegaObject3D> _object;
};
```

This example demonstrates a simple usecase where the simulation loop applies the forces to the object (see section 5.4.4), performs one simulation step (with an error check) and requests the update of the graphical object following the physical object deformation.

To create this simulation, add these lines to your program:

```
osg::ref_ptr<MyVegaWorker> worker = new MyVegaWorker(object.get(), integrator);
osg::ref_ptr<PoLAR::SimulationManager> manager = new PoLAR::SimulationManager(worker.get());
manager->launch();
```

With these lines, the simulation will start as soon as you run your program, at a framerate of 30 Hz. You can change this behaviour with the following commands:

```
// set the worker not to launch automatically
worker->setLaunchOnStart(false);

// Set the time step to 20.0 ms (so a framerate of 50 Hz)
worker->setTimeStep(20.0);
```

You can control the simulation thanks to the manager's API:

```
// run the simulation
manager->play();

//pause the simulation
manager->pause;
```

5.4.4 Adding forces to the object

You can write and add your own forces to the simulated object. When added to the object, they will then be called and applied during the simulation loop (see section 5.4.3).
To create a force, extend the base class `VegaInteraction` provided by PoLAR:

```
class ExampleForce: public PoLAR::VegaInteraction
{
 public:
   ExampleForce() {}

   ~ExampleForce()
   {
       free(f_ext);
   }

   void initializeForce(PoLAR::VegaObject3D *object)
   {
       // code of force initialization
       int numVertices = object->getVolumetricMesh()->getNumVertices();
       int r = 3 * numVertices;
       f_ext = (double*) calloc (r, sizeof(double));
   }


   double* applyForce(PoLAR::VegaObject3D *object)
   {
       // code of force computation
       return f_ext;
   }

   void resetForce(PoLAR::VegaObject3D *)
   {
       // code to reset force
   }

 protected:
   double *f_ext;
};
```

As a complete force source code can be very verbose, we provide here only the basic class outline. For a complete and detailed example of a force implementation, please refer to the example `demoVega` and more particularly to the file `VegaForces.h` which can be found in the `Examples` directory.

Once implemented, add this force to your object:

```
ExampleForce *myForce = new ExampleForce();
object->addForce(myForce);
```

Forces are deleted in the `PoLAR::VegaObject3D` destructor: you don't need to manage their destruction manually.

6 Going further

The previous sections were an introduction to PoLAR's basic features. The present section is intended to show more advanced usage possibilities.

6.1 Multiple image sequences

PoLAR manages background images stored in *sequences*. For example, medical imaging modalities often provide *sequences of images* rather than one image alone. PoLAR is able to manage sequences of images.
The example `visuseq` is an example on how to load a raw sequence of images into PoLAR, and then manipulate them.
The methods `PoLAR::Viewer::nextImageSlot()` and `PoLAR::Viewer::previousImageSlot()` can be used to travel through the sequence. Each image of the sequence can have its own list of 2D objects, or the whole sequence can share the same list. The example `demo2D` shows the possibilities of using 2D objects with a sequence of multiple images: how to copy an object from a list to another for example.

6.2 Extending 2D objects

2D objects in PoLAR are basic tools for a general usage. In some cases it can be interesting to extend their possibilities.

6.2.1 Inheriting from Object2D

When using PoLAR's helper methods for creating 2D objects, we do not need to manage the `Object2D` classes because everything is managed by the `DrawableObject2D` which has ownership of the `Object2D`. But, when access to the mathematical representation of the geometrical object is needed, it is possible to inherit from the 2D objects.

For example, if we want to define a list of control points which has more functionalities than the default `Markers2D` (e.g. to add custom methods and to overwrite the existing methods), we can simply inherit from `Markers2D`:

```
class CustomMarkers2D : public PoLAR::Markers2D
{
public:
CustomMarkers2D():
 PoLAR::Markers2D()
 {}

 //virtual methods
 void addMarker(PoLAR::Marker2D *marker);
 void removeMarker(PoLAR::Marker2D *marker);
```

The method `addMarker(PoLAR::Marker2D *marker)` is a virtual method called when a marker (i.e. a control point) is **added** to the object. The method `removeMarker(PoLAR::Marker2D *marker)` is a virtual method called when a marker is **removed** from the object.
In these methods, you need to call the superclass method first. Then, you can add any code you want to be executed when a marker is added. For example, we would like to set a custom label to set markers:

```
void addMarker(Marker2D *marker)
{
   // call the base class method first
   Markers2D::addMarker(marker);
   // then add the custom code
   setCurrentIndex(nbMarkers()-1);
}

void removeMarker(Marker2D *marker)
{
   // call the base class method first
   Markers2D::removeMarker(marker);
   // then add the custom code
   updateIndices();
}
```

For more detailed code, please refer to the `demo2D` example, and more particularly to the `TaggedMarkers2D` class defined in the `TaggedMarkers2D.h` file.

Once you have inherited from the virtual methods (if needed), you can add your custom methods.

```
  void myCustomMethod()
  {
    ...
  }

};
```

Then, pass an instance of the newly defined object to a `PoLAR::DrawableMarkers2D` which manages its graphical representation:

```
PoLAR::Viewer viewer;
...
CustomMarkers2D *myMarkers = new CustomMarkers2D();
PoLAR::DrawableMarkers2D *drawableObject = new PoLAR::DrawableMarkers2D(&viewer, myMarkers);
```

The constructor of all `PoLAR::Drawable[...]` classes takes as first parameter the `PoLAR::Viewer` on which the object will be drawn. It is indeed possible to create applications with more than one `PoLAR::Viewer` and in this case, the 2D object must know on which one it has to be drawn. The side effect is that it is impossible to create a 2D object *before* creating the viewer on which it should be drawn.
Now, to add the `PoLAR::Drawable[...]` to the viewer:

```
viewer.addObject2D(drawableObject);
```

Your object is now usable like every standard PoLAR object: you can draw it using the same keyboard and mouse interaction commands, you can access it using the same methods. To access your custom methods, you just need to cast the object into your custom type. For instance, if your object if selected, you can get it by:

```
PoLAR::DrawableMarkers2D *drawableObject = static_cast<PoLAR::DrawableMarkers2D*>
        (viewer.getCurrentObject());
CustomMarkers2D *myMarkers = static_cast<CustomMarkers2D*>(drawableObject->getObject());
myMarkers->myCustomMethod();
...
```

6.2.2 Inheriting from DrawableObject2D

Inheriting from `Object2D` provides access to the mathematical representation of the object. When you want to customize the graphical representation of the object, you will have to inherit from the `Drawable[...]` class.

Constructors

Note that in order to respect the constructor calls, it is mandatory to create at least one constructor with the first two parameters being a `std::list<Marker2D *>` and a `PoLAR::Viewer2D*`. The other parameters (if present) must have a default value:

```
DrawableTaggedMarkers2D(std::list<PoLAR::Marker2D *> list, PoLAR::Viewer2D *viewer, int
    startIndex=0):
       PoLAR::DrawableMarkers2D(list, viewer)
  {
    ...
  }
```

The `PoLAR::Viewer2D` type is a parent type of `PoLAR::Viewer` which is specialized in the computation of everything related to 2D objects.

A second constructor can be useful if the object is meant to be created manually (for example with a custom `Object2D` class):

```
DrawableTaggedMarkers2D(PoLAR::Viewer2D* viewer, CustomMarkers2D *obj):
       PoLAR::DrawableMarkers2D(viewer, static_cast<PoLAR::Markers2D*>(obj))
  {
    ...
  }
```

The call will be made like this:

```
 // create a PoLAR::Viewer
 PoLAR::Viewer viewer;
 ...
// create a custom mathematical 2D object
CustomMarkers2D *customObject = new CustomMarkers2D();

// create a custom graphical object used to draw the previously created mathematical object
DrawableTaggedMarkers2D *customDrawableObject = new DrawableTaggedMarkers2D(&viewer,
    customObject);

// set a name to the object (optional)
customDrawableObject->setName("MyCustomObject2D");

// add the new 2D object to the PoLAR::Viewer
viewer.addObject2D(customDrawableObject);

// start the edition of control points in order to draw the custom object manually
viewer.startEditMarkersSlot();
```

Virtual methods

Two interesting methods should be reimplemented in a `Drawable[...]` subclass:

- `virtual const char* className() const`
- `void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget)`

The first one is used for object serialization and must return the name of the class.

```
virtual const char* className() const
{
    return "DrawableTaggedMarkers2D";
}
```

The second method is in charge of painting the 2D item[24] using the `painter` passed in parameter. You can use it to draw your own custom shapes. Note that the drawing of the control points (*markers*) is managed by the parent class.
In this example we will use the `paint()` method to draw the marker labels next to their markers:

```
virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget
    )
{
  // call the base class method in order to draw the control points
   DrawableMarkers2D::paint(painter, option, widget);

   // get access to our custom Object2D
   TaggedMarkers2D *object = dynamic_cast<TaggedMarkers2D*>(getObject());
   if(object)
   {
     // get the list of control points
      std::list<Marker2D*> list = object->getMarkers();
      // this iterator points to the currently selected control point
      std::list<Marker2D*>::iterator current = object->getCurrentMarkerIterator();
      // select the QPen accordingly to the object state: selected or not
      QPen pen = (isSelected() && isEditable() ? getPen(DrawableObject2D::Selected) : getPen(
          DrawableObject2D::Unselected));

      // loop through the control points
      for(std::list<Marker2D*>::iterator it = list.begin(); it != list.end(); ++it)
      {
         // check if control point is the currently selected one so that we adapt our QPen
             accordingly
         if(*it == *current)
             painter->setPen(getPen(DrawableObject2D::MarkerSelected));
         else
             painter->setPen(pen);
```

[24] see the Qt QGraphicsItem class

```
            // our custom text: the label of the control point
            std::stringstream sstm;
            sstm << (*it)->label();
            double x, y;
            // get the control point coordinates from absolute reference to Qt scene reference
            int offset = 5; // this ensures the label is drawn next to the control point and not
                 on it
            transformCoord((*it)->x() + offset, (*it)->y() - offset, x, y);
            // draw the text thanks to the given painter
            painter->drawText(x, y, QString::fromStdString(sstm.str()));
          }
      }
  }
```

The expected result would be:

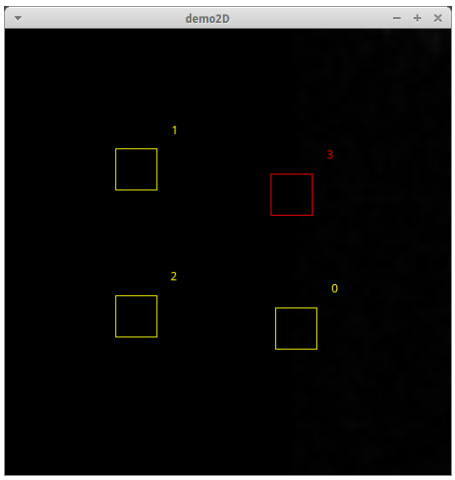

Figure 18: Custom Markers2D

Finally, in order to make your newly defined object serializable (e.g. for saving into a file. See section 6.2.4), add this macro *after* the declaration of your class:

```
// This macro ensures our custom class is registered in the serializable types
REGISTER_TYPE(DrawableTaggedMarkers2D)
```

Replace `DrawableTaggedMarkers2D` by the name of your cass.

Please see the `demo2D` example and more particularly the `DrawableTaggedMarkers2D` class which extends the `DrawableMarkers2D` class.

6.2.3 Copying an object

2D objects can be copied using different manners. All `Object2D` and `DrawableObject2D` subclasses provide a copy constructor. They also provide a `clone()` method which returns a new instance of the same type. In the case of a `Drawable[...]` object, the `clone()` method can take a boolean as parameter. Its default value is `true`, which means the newly created `Drawable[...]` will get a duplicate of the mathematical 2D object (class `Object2D` and derived). If set to `false`, the newly created `Drawable[...]` will share the same 2D object as the cloned one. That is useful when you want to share 2D objects on different images or viewers, with different graphical representations, but with the need to keep the same mathematical representation.

`Drawable[...]` objects also provide an assignment operator (`operator=`), which uses the *copy-and-swap* idiom.

```
DrawableObject2D obj1;
DrawableObject2D obj2;
...
obj1 = obj2;
```

After this assignment, the object `obj1` is equal to `obj2`, but *does not share* the same internal object. The assignment operator creates a deep copy of the object.

If you have implemented your own type of 2D objects, it is recommended that you also implement the copy constructor, the clone method and the assignement operator.

6.2.4 Saving and loading 2D objects

Human-readable format

All the default PoLAR 2D objects can be serialized and saved into a file, or loaded from a file. Custom objects can also be saved and loaded by registering them in the list of serializable types (see section 6.2.2). The objects are translated into a human-readable text format. Helper methods exist in the `PoLAR::Viewer` class:

```
void saveObject2D(const QString& fname);
void loadObject2D(const QString& fname);
```

The text format is the following:

```
<first object class name> <object name> <color when unselected> <number of control points>
<control point 1 label> <control point 1 X> <control point 1 Y>
...
<control point N label> <control point N X> <control point N Y>
<second object class name> <object name> <color when unselected> <number of control points>
<control point 1 label> <control point 1 X> <control point 1 Y>
...
.
.
.
```

Example with a blob, a spline and a polygon:

```
DrawableBlob2D myBlob #0000FF 5
0.000000 196.000000 132.500000
0.000000 84.500000 140.500000
0.000000 28.000000 98.000000
0.000000 109.500000 60.000000
0.000000 160.500000 85.000000
DrawableSpline2D mySpline #0000FF 4
0.000000 114.000000 32.000000
0.000000 82.500000 92.500000
0.000000 165.500000 162.500000
0.000000 115.000000 191.500000
DrawablePolygon2D myPolygon #0000FF 3
0.000000 160.000000 110.500000
0.000000 200.000000 183.500000
0.000000 68.500000 169.000000
```

Limitations:
As for now, it is only possible to save *all* the 2D objects present on the image. It is not possible to select which ones you want to save separately. However, it is possible to save the 2D objects of different images (in the case of image sequences - see section 6.1) in different files.
Loading from different files is however possible. The newly imported 2d objects will simply be added after the ones previouly loaded or created.
As for now, we save only the color of the object when it is unselected. In future updates we plan to add the other parameters used to draw the objects: the different colors, the shape of the control points, the thickness of the drawings, etc.

SVG export

PoLAR also provides the possibility to export 2D objects as SVG. There are different possibilities:

- export the whole scene (as a screenshot). The background image and 3d scene (if present) are saved in bitmap.
  Provided method: `void PoLAR::Viewer::saveAsSvg(QString fname)`
- export only the 2D objects, all of them at a time.
  Provided method: `void PoLAR::Viewer::saveObject2DAsSvg(QString fname)`

- export 2D objects one at a time.
  Provided method: `void PoLAR::DrawableObject2D::saveAsSvg(QString fname)`

6.3 Loading OFF models

PoLAR provides a plugin to OpenSceneGraph that allows using 3D models in OFF format. The OFF format (*Object File Format*) is used in particular by the 3D viewing program *Geomview*[25] and can be used to easily represent 3D polygonal surfaces. OFF files have the `.off` extension. Only default OFF and colored OFF (*COFF*) specifications are supported. Specifications with vertex normals or texture coordinates (*NOFF/CNOFF/STCNOFF...*) are not supported.

To let OpenSceneGraph know where the plugin is, you will need to add the path to the plugin in the environment variable `OSG_LIBRARY_PATH`.

[25] `http://www.geomview.org/`

7 FAQ

What is PoLAR?
PoLAR is a framework which aims to help creating graphical software for medical imaging and augmented reality.

What kind of help does it provide?
PoLAR offers easy binding to two heavy libraries; Qt and OpenSceneGraph. Even if it is recommended for advanced usage, it is not mandatory to know how to use Qt and OpenSceneGraph to use PoLAR.

Can I use PoLAR features without creating a viewer?
It is possible to use some features *offline* without the need of a graphical viewer, but most features are only accessible through a viewer.

Why cannot I use PoLAR to compute a projection matrix from my image?
PoLAR is designed to help creating graphical interfaces adapted to augmented reality and medical imaging. The computation of projection matrices does not correspond to what PoLAR was created for.

What is PoLAR's license?
PoLAR is released under the GNU General Public License (GNU GPL).

Whom can I contact for more information?
Please fill in the form you can find on `http://polar.inria.fr/`.

A Install OpenSceneGraph third-party dependencies from sources on Windows

The OpenSceneGraph website[26] provides Visual Studio prebuilt dependencies. But in the case the links were dead, you may need to download them from their original sources. In this example, only the `libjpeg` will be detailed, since it is the only dependency the PoLAR examples need. Applying these steps for the other dependencies should work.

To install the libjpeg dependency:

- Download the project template created by an OpenSceneGraph user: `https://github.com/bjornblissing/osg-3rdparty-cmake` (to download it, click on the `Download ZIP` button). Extract the downloaded ZIP into your `OpenSceneGraph-3.x.x` root directory.
- Download the sources for `libjpeg` from the project's website[27]. It comes as a ZIP package. Extract it into the `osg-3rdparty-cmake-master\libjpeg` directory. Be careful to extract the source files in the exact same folder as the already existing `CMakeLists.txt` file. Your folder tree should look like on figure 19.

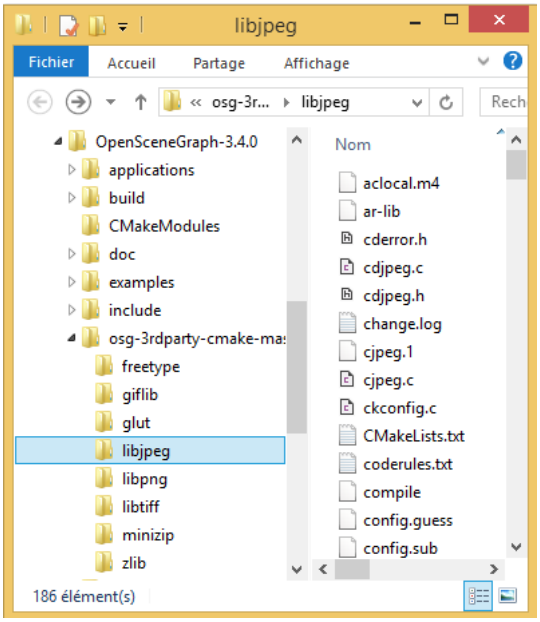

Figure 19: Directory tree of OSG dependencies

- Now launch CMake: choose the third party directory (with a default installation it should be `C:\OpenSceneGraph-3.4.0\osg-3rdparty-cmake-master`) as source code entry. To simplify the build, choose the exact same directory as output for the binaries. Click on **Configure**. Choose the compilator version you want to use; be sure to always use the same one. We recommend using Visual Studio 32-bit.
- In the red lines that appeared on screen, look for the `LIBJPEG_SOURCE_DIR` variable (you may need to tick the `Advanced` option in order to see all the variables). Set it to the `libjpeg` folder (`C:\OpenSceneGraph-3.4.0\osg-3rdparty-cmake-master\libjpeg` for a default installation) and hit `Configure` again. Once done, hit `Generate`.
- Once the generation is finished, close CMake, and open the newly created Visual Studio project (named `OpenscenegraphThirdParty.sln` in the third party dependencies folder). Be sure to select the `RELEASE` mode. You just need to hit F7 to generate the solution, and the `libjpeg` library should compile successfully.

[26] `http://www.openscenegraph.org/index.php/download-section/32-third-party`
[27] `http://www.ijg.org/`

If you need other dependencies (PNG, TIFF, etc.), proceed identically: download their sources, extract them in the right third dependencies folder, add the corresponding variables in the CMake options, configure and generate, and compile.

Now you will link these dependencies to the OpenSceneGraph project. Follow the instructions from section 2.2.2 to continue the installation.