PoLAR: a Portable Library for Augmented Reality

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

PoLAR: a Portable Library for Augmented Reality

Pierre-Jean Petitprez* Erwan Kerrien† Pierre-Frederic Villard‡

Universite de Lorraine, LORIA, UMR 7503, Vandoeuvre-les-Nancy F-54506, France
Inria, Villers-les-Nancy F-54600, France
CNRS, LORIA, UMR 7503, Vandoeuvre-les-Nancy F-54506, France

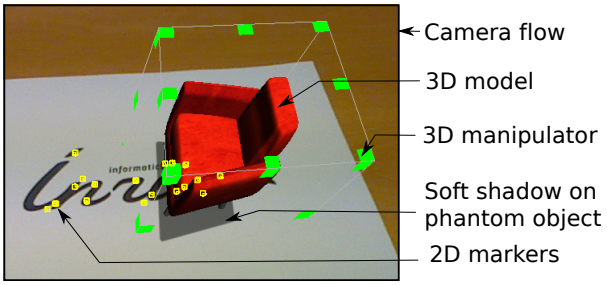


Figure 1: PoLAR tracking application showing the use of 2D objects, camera flow and real-time scenegraph interaction

ABSTRACT

We present here a novel cross-platform library to facilitate research and development applications dealing with augmented reality (AR). Features include 2D and 3D objects visualization and interaction, camera flow and image manipulation, and soft-body deformation. Our aim is to provide computer vision specialists' with tools to facilitate AR application development by providing easy and state of the art access to GUI creation, visualization and hardware management.

We demonstrate both the simplicity and the efficiency of coding AR applications through three detailed examples. PoLAR can be downloaded at `http://polar.inria.fr` and is distributed under the GPL licence.

Index Terms: 1.3.4 [Computer Graphics]: Graphics Utilities—Application packages; I.4.9 [Image Processing and Computer Vision]: Applications— [I.6.8]: Simulation and Modeling—Types of SimulationAnimation;

1 INTRODUCTION

Augmented Reality (AR) application development involves mastering many advanced topics to:

- design and implement a user friendly Graphical User Interface (GUI);
- implement an efficient graphics engine with features such as texture mapping, animation, ambient occlusion and shadow maps;
- interface various devices, especially cameras and GPS or other external tracking devices;
- implement a large diversity of Computer Vision (CV) algorithms to track moving targets, recognize objects and reconstruct scenes.

Most current AR Software Development Kits (SDK)[1] target developers with general or even little programming skills and therefore primarily provide high quality implementations of CV algorithms. ARToolkit [3] grounded its success on its powerful marker tracking and recognition, and strong camera calibration support. Vuforia[2] platform sets forward its object and scene recognition capabilities. Tracking and recognition technologies are also at the core of Metaio[3] that addressed customers with absolutely no programming skill through its Creator drag and drop application creation tool. Moreover, the latter two are commercial SDKs. This focus misses CV specialists' whose aim in developing AR applications is to test or demonstrate their own CV algorithms.

PoLAR (Portable Library for Augmented Reality) is motivated by offering those latter developers an easy access to state of the art GUI creation, image interaction, graphics engine implementation and camera interfaces, in a few lines of code, portable across both desktop and mobile platforms through an API designed to meet their needs. PoLAR defines a framework for simple and fast prototyping of graphical applications for augmented reality, and more general image visualization, in both classical CV and Medical Imaging contexts. No tracking or other image processing capabilities are offered, but instead an API to easily plug in custom tracking processing. The framework is written in C++ and published under the GNU GPL license. PoLAR can be downloaded at `http://polar.inria.fr`.

PoLAR's main contributions are i) portability: PoLAR has been tested and used on Linux, Windows, MacOS and Android; ii) simplicity: an API designed for CV specialists with detailed user guide including tutorials and examples; iii) deformation capability: PoLAR provides an API to manage physics engines; iv) sustainability: PoLAR depends on well-established Qt and OpenSceneGraph libraries, and is open source software.

2 METHODS

Supporting libraries

PoLAR depends on two main libraries: the GUI-oriented framework Qt[4] and the graphics engine OpenSceneGraph[5]. Qt allows for fast GUI development, powerful 2D graphical drawings, and high-level, cross-platform, access to hardware components, including cameras. Besides, OpenSceneGraph is a powerful 3D graphics toolkit, compatible with OpenGL[6] on desktop environments, and OpenGL ES on mobile platforms. Its scenegraph implementation enables graph nodes to be shared between multiple views, dynamically loaded or deleted, and manipulated interactively.

*e-mail:pj.petitprez@gmail.com
†e-mail:erwan.kerrien@inria.fr
‡e-mail:pierrefrederic.villard@loria.fr

[1] `http://socialcompare.com/fr/comparison/augmented-reality-sdks`
[2] `http://www.vuforia.com`
[3] `https://en.wikipedia.org/wiki/Metaio`
[4] `http://www.qt.io/`
[5] `http://www.openscenegraph.org/`
[6] `https://www.opengl.org/`

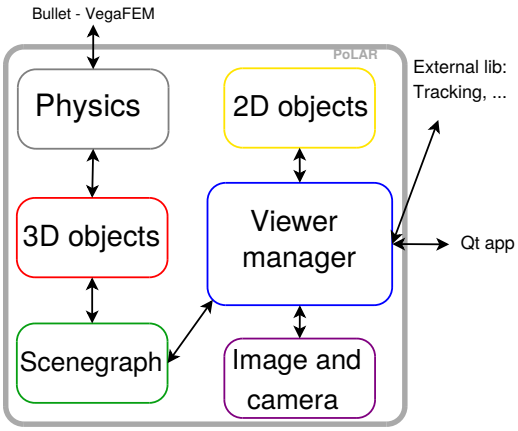


Figure 2: Library components and dependencies

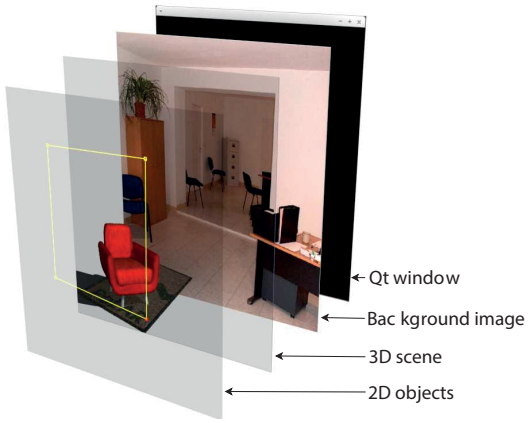


Figure 3: Composition of a PoLAR application

Both libraries were chosen for their sustainable development framework. Middle to long term support is indeed grounded on: their portability since both are available on Linux, Windows, Mac OS and Android, their open source licence for non-commercial usage and their compliance with the GNU GPL, their very active community of users and developers, their active development state with frequent updates.

Software architecture

PoLAR encapsulates calls to Qt and OpenSceneGraph in a normalized API that makes it straightforward to implement classical features required in AR application development. Yet, particular care was taken in keeping the API fully compliant with both Qt and OpenSceneGraph coding standards so that any advanced developer can still benefit from the full APIs of these libraries.

The main class of PoLAR (Viewer) inherits from a Qt widget, leading to an easy and straightforward integration of PoLAR into any Qt-based application. This widget manages an OpenSceneGraph viewer with one orthographic camera to enable pan, zoom and window-level capabilities on the display, and a perspective camera to render the 3D scene. Qt 2D graphical objets are rendered as overlay. Fig. 2 shows the various main classes of PoLAR and its links to external libraries. Fig. 3 depicts the different layers that compose a PoLAR Viewer. We briefly focus on the design of these layers.

2D graphical objects Many existing libraries (e.g. Cairo[7], GTK+[8], SDL[9] , Qt) already offer powerful 2D drawing tools that generally require to understand the notion of a graphical context, or to manage by oneself the painting events, or even to implement the mathematical representations of the needed effects. PoLAR is targeted for users who prefer to avoid dealing with such technical aspects. PoLAR wraps the Qt graphics items into a level of abstraction where paint events and drawing methods are hidden to the user, though still reachable when needed.

The creation of a new 2D object (polygon, spline, marker points (see Fig. 1) or text) takes only one line of code with default options, and up to three to set some particular options (including color, size or shape). For comparison, Qt-native objects require dealing with the paint event method and need to be manually added to the viewer, for a total of a few tens of lines. New classes of 2D objects can be implemented through a factory mechanism.

[7] https://cairographics.org
[8] http://www.gtk.org
[9] https://www.libsdl.org

Real-time interaction with a complex scenegraph PoLAR supports user interaction thanks to the powerful Qt interaction management. Default interaction methods are implemented and users can add theirs. Interaction with images, 2D objects, and 3D objects is made possible in real-time within a running application.

Multiple helper classes are provided to create and manipulate 3D objects. Each 3D object is added as a node of a scene graph which allows for fast access and manipulation of the objects in a very transparent and simple way. AR-oriented features are implemented in PoLAR, including easy access to scene lights and cameras, facilitated replacement and deletion of objects, rapid switch of objects visibility or state (normal vs phantom), phantom objects to handle occlusions of virtual objects by real objects (see the virtual chair occluded by the real table in Fig. 4, right), and high quality shadow casting. Two lines of code are needed to add a manipulator to a 3D object and use it for translating, rotating or scaling (see Fig. 1, which makes PoLAR particularly suitable for WYSIWYG applications. Interaction with a 3D object can also be made programmatically, as PoLAR offers all the methods needed to access the graph nodes.

Camera flow and images PoLAR offers tools to manipulate textures, from bitmap textures loaded from a file, to dynamic textures displaying a camera flow. Texture creation from a file or a video flow is made as simple as possible. This abstract level of management allows for texture manipulation regardless of its nature. In only two lines of code (three in the case of a dynamic texture), a texture can be loaded and applied to a 3D object, or displayed in full size in the viewer, just like in an image editor. PoLAR provides basic image manipulation features particularly suitable for medical imaging applications: pan and zoom for easy navigation on the image, and window/level interactive contrast enhancement tool.

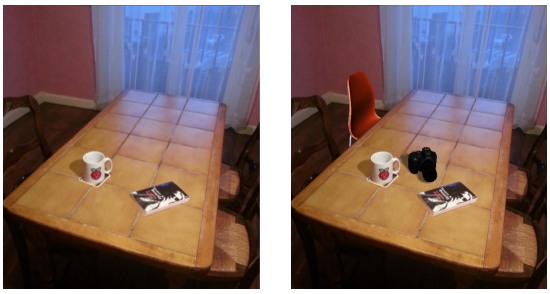
Portability

We have designed PoLAR to be compatible with all current major platforms, namely Linux-based distributions, Microsoft Windows, MacOS and Android systems. Support for Apple iOS is currently under development. This has been possible thanks to the availability of Qt and OpenSceneGraph on said platforms, and thanks to the use of C++11-compliant standard code.

All interaction methods and shortcuts are fully customizable though default values are always provided. Mouse and keyboard are the main interaction methods on desktop environments, while on mobile systems Qt-based gesture recognition allows interacting through the touchscreen.

GPU computing

We have developed a fully functional shader system compatible



Raw video flow Augmented video flow

Figure 4: Basic AR application: image without and with augmentation

with both desktop OpenGL and OpenGL ES specifications. It offers a proper per-pixel Phong reflection model [4] to objects added in the virtual scene. This shader system takes advantage of the OpenSceneGraph shadow computation features to provide soft shadows in the scene (see Fig. 1). On older desktop environments which are only compatible with previous OpenGL specifications, it is still possible to use a fixed-function rendering which uses a per-vertex Blinn-Phong lighting model [1].

Deformations

AR applications in the medical context often need to include deformations. Various methods exist to simulate deformations. They often consist of a trade off between fast and accurate simulation. The mass-spring method [2] is a real-time method consisting of modeling soft bodies as a discrete system of mass points linked together with springs and dampers. The finite element method [9] is a numerical technique for solving mechanical equations (conservation laws, balance equations, constitutive laws, etc.) considering the soft bodies as a continuum through a matrix representation.

We propose to offer both capabilities by interfacing our library with two physical engine libraries: 1) a mass-spring system based on Bullet[10] and 2) a finite element method based on VegaFEM [7]. Our strategy was to factorize the common operations linked to the physics engine (apply boundary conditions, define mechanical parameters, iterate in the system resolution of the physical laws, etc.) as well as to use our core library functionalities (display and position of a 3D mesh in an AR environment). Furthermore, we optimized the object deformation display process such as, if the topology remains constant, the mesh vertex positions are directly updated by references to the physical engine data.

PoLAR relies on the robust Qt threading API to provide multithreading computation. The chosen architecture ensures the physics computations do not interfere with the GUI and the graphical rendering.

3 Results

The main outcome of using PoLAR is a simple and efficient way of implementing AR applications. We present in this result section three kinds of application examples that highlight the code simplicity as well as the possibilities of our library.

Basic AR application

The first result example (whose source code is available in the PoLAR repository) is a basic AR application consisting of reading a camera flow, defining a projection matrix **M** and displaying 3D objects projected on the video flow with **M**. The result is displayed on Fig. 4. The initial camera image is on Fig. 4.*left* and the augmented image is on Fig. 4.*right*. A phantom object (*the table*) is used to hide a part of a 3D object (*the orange chair*) and to allow another 3D object (*the camera*) to cast a shadow on *the table*.

[10] `http://bulletphysics.org/`

Listing 1 shows the code corresponding to this application. *L4* creates the Qt application. *L5-7* create a viewer with light and shadows. *L8-13* add video stream (a webcam on `/dev/video0`) on the viewer background. A video manager is created, then a texture from it and then this texture is displayed on the viewer background. *L14* enables pan and zoom and window/level on the display. *L15-21* set up the 3D scene: *L15-16* load and add a first 3D object (*the camera*) *L17-18* load and add a second 3D object (*the orange chair*), and *L19-21* create a ground plane (*the table*), set it as a phantom object and add it to the scene graph. With *L22-27*, every time a new frame is captured, the video manager sends a Qt signal with the image data and connect it to a custom Qt slot function which computes the related projection matrix. In this example a lambda function is used (*L23-27*) which should be completed on *L24* with the code of a custom tracking algorithm that updates the projection matrix **M**. *L26* applies the computed projection matrix to the viewer. Conversion from vision-oriented matrices to OpenGL-oriented matrices is managed by PoLAR. *L28* shows the widget, *L29-30* ensures the application correctly quits when the viewer is closed, and runs the application.

Listing 1: Basic AR application code

```
1  typedef osg::ref_ptr<PoLAR::Object3D> Obj3Dptr
2  int main(int argc, char** argv)
3  {
4      QApplication app(argc, argv);
5      PoLAR::Viewer viewer;
6      viewer.setShadowsOn();
7      viewer.addLightSource(5,-5, 5.5, true);
8      osg::ref_ptr<PoLAR::VideoPlayer> videoStream
           = new PoLAR::VideoPlayer(0);
9      osg::ref_ptr<PoLAR::Image_uc> myImage = new
           PoLAR::Image_uc(videoStream);
10     viewer.setBgImage(myImage);
11     viewer.bgImageOn();
12     videoStream->play();
13     viewer.setResizeOnResolutionChanged();
14     viewer.startEditImageSlot();
15     Obj3Dptr obj = new PoLAR::Object3D("data/
           reflex_camera.obj");
16     viewer.addObject3D(obj);
17     Obj3Dptr obj2 = new PoLAR::Object3D("data/
           plastic_chair.obj");
18     viewer.addObject3D(obj2);
19     Obj3Dptr ground = new PoLAR::Object3D("data/
           table.obj");
20     ground->setPhantomOn();
21     viewer.addObject3D(ground);
22     QObject::connect(videoStream.get(), &PoLAR::
           VideoPlayer::newFrame, [=](unsigned
           char* data, int w, int h, int d)
23     {
24         /* tracking code here */
25         osg::Matrix M = /* set the computed
               projection matrix */;
26         viewer.setProjection(M);
27     });
28     viewer.show();
29     app.connect(&app, SIGNAL(lastWindowClosed())
           , &app, SLOT(quit()));
30     return app.exec();
31 }
```

This first example shows how the core visualization tools have been encapsulated. Writing the same application using a 3D visualization library such as OpenGL or out-of-the-box OpenSceneGraph would have been cumbersome and many functionalities would be missing (e.g.: window/level, webcam management).

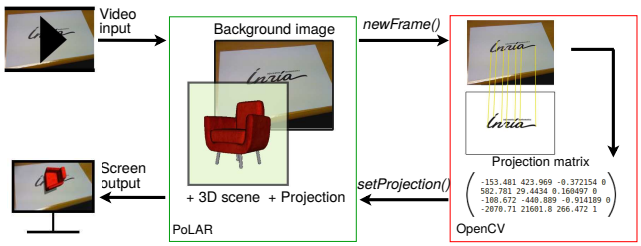


Figure 5: Workflow of the advanced AR application

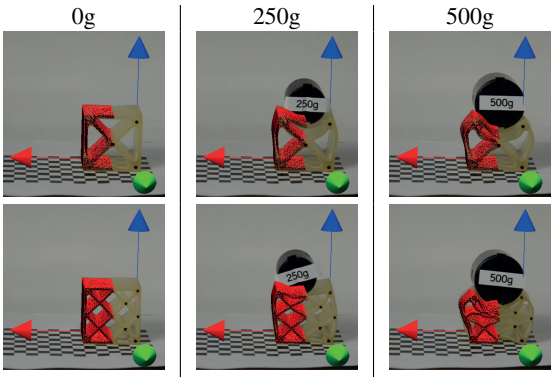


Figure 6: Real and virtual cubes deformed under various weight. Two shapes were tested with no weight, 250g and 500g. Half of the virtual cube is displayed to better appreciate the deformed shape.

Advanced AR application including tracking

As studied in the previous application example, PoLAR aims to facilitate the creation of graphical applications, and does not provide any projection matrix update method, but provides suitable inputs and outputs to link it to e.g. a tracking library. To validate the efficiency of PoLAR in real use cases, we have developed an AR application (the source code is available in the PoLAR repository, under the `runPolar` application) where PoLAR is bound to the OpenCV library [5] to implement the tracking of an image marker. PoLAR API defines two functions to implement this binding: `newFrame` sends the current camera frame to the tracking algorithm, and `setProjection` applies the computed projection matrix to the 3D scene.

Thanks to these very generic methods, it is very easy to replace the current OpenCV tracking algorithm by any other tracking method either user-defined or available from another tracking library. Fig. 5 shows the workflow corresponding to the code interaction between the PoLAR-based class and OpenCV tools.

AR application with deformations

Finaly, we tested the VegaFEM-based deformation module by building physical objects that can significantly deform, i.e. the deflection could easily be observed by human eyes. These objects are 3D-printed cubic-shaped objects with a very soft polymer material. The experiment consisted of placing a cube at the origin corner of a chessboard and applying different weights (250g and 500g) on it, taking a picture at every step, including the rest state. The same experiment was reproduced within PoLAR. The projection matrix was computed via the camera calibration method presented in [8] and provided by OpenCV. The initial 3D models without deformation and projected on the pictures can be seen on the left of Fig. 6. The deformation simulation was done using the PoLAR physics plugin. The cubes were meshed with tetrahedrons using tetgen [6]. The boundary conditions were a Dirichlet boundary condition (zero displacement) applied on the bottom of the cube complemented by a Neumann condition that imposed a force on the top of the cubes to reproduce the weight. The integration was done using the implicit Newmark scheme with a time step of 0.02 and the two Newmark constants were $\beta = 0.25$ and $\gamma = 0.5$. The corotational formulation was used to provide with large and stable deformations. The material was defined with a linear constitutive law characterized by a Young's modulus around 1 MPa found by trial-and-errors and a Poisson's ratio of 0.49 to set the material incompressible.

Our tests consisted of comparing the real experiment and the PoLAR-based simulation. The deformations were observed in both virtual and real worlds. Two cubes with different cross-section shapes were tested. Results are illustrated on Fig. 6. In order to better appreciate how the simulation matched with the reality, half of the virtual cube has been hidden. Qualitatively, one can observe that it is feasible to reproduce both the pose and the deformation in an augmented environment. Quantitatively, the outcome was twofold: *Coding simplicity*: we implemented this application in approximatively 200 lines, with 150 lines required by the simulation with VegaFEM, and only 50 lines for the full display using PoLAR. This is due to the higher abstraction level in both the visualization and the deformation functions. ii) *Efficiency*: our simulation runs at 20 fps. This experiment is available on our GIT server.

4 Conclusion

We have presented PoLAR, a new open-source library that helps with building AR applications. It contains a set of tools to display images or video flows, to add 2D and 3D objects via a scenegraph environment with 3D deformation capabilities.

PoLAR is cross-platform (MacOS, Linux, Windows and Android), takes advantage of modern GPU capabilities, handles common hardware constraints (e.g.: camera flow and tactile device interactions) and provides with real-time applications.

PoLAR has been tested on many examples and basic applications all available on the website `http://polar.inria.fr`. Three of them are described in this paper to show the simplicity and the efficiency of coding AR applications compared to other available tools: low number of lines of code, self-explanatory methods, fast computing time, interaction simplicity with external libraries and virtual soft-body deformation simulation.

Acknowledgements

PoLAR was funded by an Inria Tecnhnology Development Action grant (2014-2016).

References

[1] J. F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[2] M. Desbrun and A. Barr. Interactive animation of structured deformable objects. In *Proceedings of Graphics Interface (GI)*, pages 1–8, 1999.

[3] H. Kato, K. Tachibana, M. Billinghurst, and M. Grafe. A registration method based on texture tracking using artoolkit. In *Augmented Reality Toolkit Workshop, 2003. IEEE International*, pages 77–85, Oct 2003.

[4] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[5] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. Realtime computer vision with opencv. *Queue*, 10(4):40:40–40:56, Apr. 2012.

[6] H. Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2):11:1–11:36, Feb. 2015.

[7] F. S. Sin, D. Schroeder, and J. Barbic. Vega: Non-Linear FEM Deformable Object Simulator. *Computer Graphics Forum*, 2013.

[8] Z. Zhang. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(11):1330–1334, Nov. 2000.

[9] O. Zienkiewicz, R. Taylor, and J. Zhu. In *The Finite Element Method Set*. Butterworth-Heinemann, Oxford, sixth edition edition, 2005.