# Using the GeoX Framework


*April 16th, 2014*

## 1. Introduction

GeoX is a collection of C++ libraries for experimenting with geometric modeling and visualization techniques (GeoX = geometry experiments). It consists of three major parts that help working with geometric problems. The first part is an object oriented library for structural reflection that allows for rapid application prototyping. The second part is a library that provides basic mathematical tools such as vectors, matrices and solvers for linear systems. The third part is a system to conduct geometric experiments that provides viewers for 2D and 3D geometry.

The original version of GeoX has been developed by Michael Wand and is based on the open source XGRT system:
http://www. mpi-inf.mpg.de/~mwand/XGRT/index.html

## 2. Object-oriented base system and structural reflection

The main feature of the system is a system for *structural reflection* (also called *introspection*). Structural reflection means that the runtime system of a program knows structural information about how the program has been structured internally. In our context, this refers to classes, their inheritance hierarchy and class members such as variables and methods. The GeoX system provides such a mechanism, which is usually not available to C++ programs (other languages such as Java or .net-compatible languages provide structural reflection as part of the standard system libraries). Using structural reflection, the system can in particular perform two important tasks:

- Automatically create a simple user interface for editing instances of classes
- Load and safe objects from/to disk

In both cases, the system inspects the class for its properties (member variables), and either displays an editor for each property or read/writes the property for file storage. In order to use this mechanism, three steps are necessary:

(1) Declare a class as a GeoX-class

(2) Register all relevant members

(3) Register the class with the system

An example with detailed comments can be found in folder `experiments/ExampleExperiment.h`, `experiments/ExampleExperiment.cpp`. To declare a class as GeoX-class, an additional macro has to be added to the class declaration. The header file will look like this:

```
#include "Persistent.h"

class Example : public Persistent
{
   GEOX_CLASS(Example)
 private:
   // private members
```

```
 public:
   // public members
}
```

The class must be directly or indirectly derived from "Object"; for storage on disc, it must be derived (directly or indirectly) from "Persistent". The "Experiment"-class (see section 4) is a descendant of Persistent. Hint: If your class is abstract, use the macro "GEOX_ABSTRACT_CLASS(Example)".

The CPP-file then has to contain another macro:

```
#include "Example.h"

IMPLEMENT_GEOX_CLASS( Example, 0 )
{
   BEGIN_CLASS_INIT( Example );
}
```

The number "0" refers to the class version, starting at zero. If properties are added later, this number has to be increase (along with the version numbers of the newly introduced properties) so that old files can still be read[1]. Again, if you are defining an abstract class, use IMPLEMENT_GEOX_ABSTRACT_CLASS.

In order to add properties, you have to add one line with a macro to your cpp file for each property. Here is an example. Let's assume the example class has the following members:

```
#include "Persistent.h"

class Example : public Persistent
{
   GEOX_CLASS(Example)
 private:
   int32 a;
   card32 b;
   Vector3f c;
   void doSomething();
}
```

The types (int32, card32) are defined in PTypes.h; these are aliases to standard C++ types that have a well defined meaning across different platforms (win32, win64, linux32, linux64). Vector3f is a math type discussed below. To register the properties, add the following macros to your cpp file:

```
#include "Example.h"

IMPLEMENT_GEOX_CLASS( Example, 0 )
{
   BEGIN_CLASS_INIT( Example );
   ADD_INT32_PROP(a, 0)
   ADD_CARD32_PROP(b, 0)
   ADD_VECTOR3F_PROP(c, 0)
}
```

The following properties types can be registered:

- Numerical values (see NumericalClassProperty.h): int8, int16, in32, int64, card8, card16, card32, card64 (unsigned ints), float32, float64.
- Vector and matrix types: Vector2f, Vector3f, Vector4f, Matrix2f, Matrix3f, … (see FixedArrayClassProperty.h)
- Strings: see StringClassProperty.h
- Pointers to GeoX objects: ADD_OBJECT_PROP(name, version, BaseClass::getClass(), owner). name is the name of the variable, version is the version number, BaseClass is the name of the class the registered pointer's class is derived from, owner is true if the current class owns the pointer (i.e. deletes it in its destructor), false otherwise.

[1] Removing properties is possible but more complicated and involves redefining the read and write methods; this is not discussed here.

- A few more (see "properties" directory in "system")
- experiments/ExampleExperiment.h shows an example.

In addition, it is also possible to register methods. Methods are limited to those that do not have parameters and do not return anything. Such methods will appear as buttons that can be clicked by the user in order to call the method. In order to register a method, use the macro "ADD_NOARGS_METHOD()". As parameter to the macro, specify the (fully qualified) name of the method, so for example "Example::doSomething".

Class registration: Finally, after declaring a new GeoX class, you need to let the system know that it exists. For this purpose, the file system/basics/InitGeoX.cpp is used: In this file, all classes have to be registered. There are two functions, init() and shutdown(); one is called when the system starts, the other when the system is shutting down. Register your class in three steps:

- Add an #include directive to provide the header of your new class.
- In function init(), call YourClass::init(BaseClass::getClass()) to register the class. Make sure to specify the correct base class so that the introspection mechanism will find your class at the correct place.
- In function shutdown(), call YourClass::shutdown()

## 3. Math tools

GeoX comes with a set of mathematical tools in the "math" directory. For vectors and matrices of small, fixed dimension, the module LinearAlgebra.h can be used (in case you use this, do not forget to include "LinearAlgebra.hpp" in your cpp files; this file contains all the inline and template code. Most C++ compilers need to have this implementation at hand when compiling template/inline code). The library provides the classes StaticVector<FloatType, dim> and StaticMatrix<FloatType, dim>, which provide vectors and matrices of fixed size and of any floating point type (both are template parameters). There are predefined types Vector3f for three dimensional float vectors, Vector2i for two-dimensional int vectors and Vector4d for four dimensional double vectors, and all similar types as well. The library also defines Matrix2f, 3f, 2d,3d,4d … and so on. Operator overloading is used to provide all familiar matrix and vector operations; the function "invertMatrix()" computes inverse matrices using Gaussian elimination.

A similar library exists with the name DynamicLinearAlgebra.h|hpp|cpp. This library provides the same functionality, but the dimension of the matrices and vectors can be specified at runtime (not a template argument).

The library "SparseLinearAlgebra.h|hpp|cpp" provides sparse matrices. Please note that these matrices are stored row-wise, while all other matrices are stored column wise. This is necessary to support efficient multiplications of columns vectors with sparse matrices. Efficient solvers for sparse linear systems (including eigenvector computations) are located in "IterativeSolvers.h"

## 4. Experiments

The main GeoX application shows an experiment and a viewer. You can change to different experiments using the drop-down box at the top-right. This drop down box shows (by introspection) all classes derived from "Experiment". Make sure to derive a new experiment from this base class so that it appears in the box. The area on the right will then show all methods and properties of your experiment, and the area on the left will show a viewer.

To specify a viewer, override the method createViewer() (see experiments/Experiment.h). This method is called whenever switching to a different experiment to create a new viewer. The method should

maintain a copy of a pointer to the viewer for future reference. Through this pointer, you will be able to draw different geometric objects in the viewer.

Three viewers are available:

- A simple viewer that just shows a logo (not really helpful).
- A 2D viewer can be found in `viewers/widgets/ GLGeometryViewer.h`. The methods are self-explanatory – you can basically add points and lines with different colors.
- A 3D viewer, located at `viewers/widgets/GLGeometry Viewer3D.h`.

In addition to the viewer, you can at any time output text to the console at the lower bottom of the window using the "`output`" object (include `GeoXOutput.h` from `system/gui`). It uses the familiar <<-notation similar to the standard C++ library.

For a complete example of an experiment with and without viewers, please refer to the example code in `experiments/ExampleExperiment.h` and `experiments/ExampleExperiment2DGraphics.h`.

## 5. Installation and Configuration

In order to build and use GeoX, you need:

A supported C++ compiler:

- **Windows:** We have tested GeoX with Microsoft Visual Studio 2008 and 2010 (separate solutions provided; look for the GeoX*.sln files in the "windows" folder). These projects create a 32-bit application. Therefore, they also need a 32-bit Qt. The freely available "express" versions of MS Visual Studio work fine for this purpose.
- **All Platforms:** We provide a CMake build script as well. It should be easy to bring GeoX to any platform using CMake, including Windows and Linux.

An installation of Qt Version 4.x (tested with 4.8.5).

- Qt is freely available for download at: `http://qt-project.org/`
- You need to make sure that the qt bin, include and lib directory are within the search paths of your build environment. For MS Visual Studio, this is setup in the Options/Projects And Solutions/Project Directories Menu (VS 2008) or in the project settings: Project/Properties (VS 2010)/Configuration Properties/VC++ Directories.
- In case you are using Visual Studio, make sure to setup these right after installing QT (Build does not work otherwise). For VS2010, please note that you need to load the project first to be able to make the settings (which are in turn stored in the project file). The standard settings for VS 2010 use the environment variable %QTDIR% which should work most of the time, but it is better to check. For VS 2008, standard settings cannot be provided.

**Troubleshooting / FAQ:**

- The program does not compile because qt includes are not found
  → make sure that the qt "include" directory is in the include path of your development environment
  → check whether the right directory is referenced there (not subdir or parent dir)
- The program does not link because of missing QT dlls
  → make sure that the qt "lib" directory is in the library path of your development environment
- The program does not start because QT DLLs are not found (Windows)
  → Either adapt system binary search path %path% to include QT's bin directory
  → Or: copy the QT DLLs into the build directory. For the "debug" configuration, you will need `QtCored4.dll`, `QtGuid4.dll`, and `QtOpenGLd4`. For the "release" configuration, you will need `QtCore4.dll`, `QtGui4.dll`, and `QtOpenGL4` (no "d" in the DLL name).
- When I create a new viewer, the code does not compile/link because of missing "Meta Object" related functions.
  → Make sure that the Meta-Object-Compiler (moc) is run on the header file of your new code. Visual Studio: Set a custom build rule for this header file (Solution Explorer, right click on header, Properties, Custom Build Steps)
  Other platforms: check the .pro file.
  → Make sure you include the generated code in your project
  → Maker sure that the class has the Q_OBJECT macro on the first line after "`class xxx: public yyy {…`"