Using the GeoXL Framework

February 25th, 2013

1. Introduction

GeoX is a collection of C++ libraries for experimenting with geometric modeling techniques (GeoX = geometry experiments). GeoXL is an extended version that includes several tools for point cloud processing.

GeoXL 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 experiment that provides viewers classes for 2D and 3D geometry. GeoX 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 GeoXL 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 GeoXL-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 GeoXL-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 {
   GEOXL_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 “`PCIDGPExperiment`”-class (see section 4) is an indirect descendant of `Persistent`.

Hint: If your class is abstract, use the macro “`GEOXL_ABSTRACT_CLASS(Example)`”.

The CPP-file then has to contain another macro:

```
#include "Example.h"

IMPLEMENT_GEOXL_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_GEOXL_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 {
   GEOXL_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_GEOXL_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`)

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

- Strings: see `StringClassProperty.h`
- Pointers to GeoXL 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.
- 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 GeoXL class, you need to let the system know that it exists. For this purpose, the file `system/basics/InitGeoXL.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

GeoXL 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. Point Clouds and Experiments

Point clouds are handle by the class "`PointSet`": A PointSet is a 2D-array of points (the second dimension is used for depth images such as the results from a kinect capture; for unstructured point clouds, always set the width to 1; this gives a 1D array of points).

The points in a point set can have different channels. Each channel can have one out of three basic types (card8, int32, float32) and an arbitrary dimension. For example, a "position" channel would have three floats, encoding a position in three-space.

The channels are set at *runtime*. The channels are constant over all points. Channels are controlled by a class called "`VertexDescriptor`". For example, in order to create a point set with position and color channel, use the following code:

```
#include "PointSet.h"

...

PointSet *ps = new PointSet();

const int numPts = 1000;

VertexDescriptor vd;

vd.pushAttrib(mVAD("position", 3, VAD::DATA_FORMAT_FLOAT32));

vd.pushAttrib(mVAD("color",    3, VAD::DATA_FORMAT_FLOAT32));

ps->clearAndSetup(1, numPts, &vd);}
```

There are a number of standard channel that almost all point clouds will/should have:

name	type	dimension	description
position	float32	3	world coordinate of the points in the point cloud
color	float32	3	color as RGB
flags	int32	1	flags for individual points (see PointFlags.h) most important: FLAG_INVALID means that the point should not be considered (for example for empty pixels in a depth image)
normal	float32	3	optional: normal orientation for shading

In order to access channels, you have to create an "`AttributeAccessToken`" or "`AAT`" (typdef'ed). For example, for reading all positions of a point clouds, use:

```
#include "PointSet.h"

...
PointSet *input = ...;

mpcard numPts = input->getNumEntries();

AAT positionAAT = input->getAAT("position");

// alternative: input->getAAT("position", 3, VAD::DATA_FORMAT_FLOAT32);

for (mpcard i=0; i<numPts; i++) {

   Vector3f pos = input->get3f(i, positionAAT); // read

   input->set3f(i, positionAAT, makeVector3f(1,2,3)); // write,

}
```

PointSets are packaged into larger structures, called PointClouds. These are then packaged into nodes in a scene graph. In order to facilitate the handling of these elements, we provide a class "PCIDGPExperiment" in the module "geoxlKinnect". It abstracts from the more complex system environment.

The class is a "PointCloudInteractionTool", which is provides means to interactively work with a scene of point clouds. You can declare properties and methods (with buttons) as described above to define your experiment and apply it to the scene.

As with any GeoXL class, you have to register your experiment to work. Usually, you should register in "InitGeoXLKinect" (and add all of your code to the GeoXLKinect project). If you use a different project, use the right init file and do not forget to add the correct calling convention when declaring the class (for GeoXKinect this is "GEOXKINECT_API").

5. Installation and Configuration

In order to build and use GeoXL, you need:

A supported C++ compiler:

- **Windows:** We have tested GeoXL with Microsoft Visual Studio 2008 and 2010 (separate solutions provided; look for the GeoXL.sln and GeoXL.vs2010.sln in the "windows" folder).
- An installation of Qt Version 4.x (tested on 4.8.1).
- Qt is available for free for download at: `http://qt.digia.com/`
- After installing QT, you should add the following environment variables to your system (right click on "computer" symbol in control panel or on the desktop, then chose properties, advanced settings (windows 7), environment variables, then add user or system variable):
  %QTDIR32% should be pointing to the 32bit version of QT.
  %QTDIR64% should be pointing to the 64bit version of QT.
  Please set the directories such that %QTDIR32%\include, %QTDIR32%\lib, %QTDIR32%\bin exist and point to the corresponding directories of the QT SDK for includes, libraries and binaries.
  Attention: these variables are only observed by GeoXL, they are not set

automatically by the QT installation (the reason is that we need to distinguish 32/64bit, which the automatic installer cannot handle).
Once this is set, everything should work out of the box.

- For Kinect support, install the Kinect for Windows SDK and Developer Toolkit as explained on the Microsoft web pages. Please note the mandatory installation order for these two components.
(http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx).
Afterwards, install the OpenNI SDK (http://www.openni.org/). Make sure to chose the right version for your operating system (32/64bit). If you want to build the other configuration as well, install this, too (but it will not execute correctly on a mismatching platform).

Troubleshooting / FAQ:

- The program does not compile because qt includes are not found
or the program does not link because of missing QT dlls
→ make sure that the QTDIR32/64 variables are set
→ check whether the right directory is referenced there (not subdir or parent dir)
- The program does not start because QT DLLs are not found
→ make sure that the QTDIR32/64 variables are set
→ Worst case: copy the QT DLLs into the build directory. For the "debug" configuration, you will need `QtCored4.dll`, `QtGuid4.dll`, `QTXMLd4.dll`, and `QtOpenGLd4`. For the "release" configuration, you will need `QtCore4.dll`, `QtGui4.dll`, `QTXML4.dll`, and `QtOpenGL4` (no "d" in the DLL name).
- My new Experiment does not show up.
→ Check if you have registered it in "initGeoXLKinect" (or the appropriate subproject).
- OpenNI does not work.
→ Check the operating system version and the enviroment variables
→ Attention: Environment variables are only set when Visual Studio starts. After changes, restart VisualStudio.