CGoGN plug

A quick SOCIC2012 project overview
+
how to use

Hurstel Alexandre

November 5, 2012

Contents

1 Introduction and purposes

1.1 CGoGN

1.1.1 What is CGoGN library ?

CGoGN is a C++ topological geometric modeling kernel that provides efficient and generic data structures based on n-dimensional combinatorial maps.

The main purpose of **CGoGN** is to provide a strong separation between the description of the topology of the mesh (the cells and their adjacency relationships) and the space in which the cells are embedded (attributes associated to the cells) in the meantime the library also provides efficiency (more contiguous tables, less pointers), low memory consumption, genericity (models, dimensions).

A good start to get familiar with **CGoGN** and all of its features is to get familiar with topology and check the **CGoGN** project page.

1.1.2 CGoGN and visualization?

Since **CGoGN** is a topological library, it has to provide graphical tools to visualize the topological models it allows to generate although it's not quite its priority concern.

That's why the **CGoGN** library provides some useful tools of visualization that take forms of C++ classes that are actually inheritance or implementation of **Qt** framework's C++ classes. Within the same state of mind, the library also provides easy to use C++ rendering functions and classes that operate several **OpenGL** features and functions.

1.2 The project

1.2.1 CGoGN in space?

The aim of this project is not to replace or improve any of the **CGoGN** rendering classes of functions, but to use these provided features to create and implement a new approach to provide visualization for **CGoGN**'s objects.

With the current GUI tools provided by **CGoGN**, the programmer using the library generally has to inherit some classes that are themselves inheritance of some **Qt** GUI classes (see the SimpleQt example). Doing so, he will create a new and independent

application. The main idea of the project is to allow the coder, not to create his own application, but to write code and add it dynamically to a main application, already containing basics GUI features.

1.2.2 Plugins and CGoGN?

In order to offer such a system, the goal of the project was to provide CGoGN with an application that integrates a **plugin system**. This means that any coder could develop is own third party code as a compatible plugin and add it to the main application as a new feature, without ever touching the main application's inner code or even without having to recompile it.

The function of the main application is to provide visualization (with several functionalities) of CGoGN generated objects combined with a few possibilities of GUI to interact with these. However, the application itself, without any plugin attached to it, doesn't give any visualization or relevant interface. Although the application provide this functionalities, they have to be activated and called by the plugins that will be attached to the application. As a metaphor, the main application is like an empty house with electricity and running water, but that has to be filled with various furniture, appliance, decoration, etc...

2 The project

2.1 Work

2.1.1 Specifications

The project was developed to meet several specifications:

- The plugins can provide a drawing function for the application to show
- Several cameras can show at the same time a 3D visualization from a different point of view.
- The application can show several different visualization at the same time.
- A plugin draw function can complete another plugin's draw function.
- The plugins can add entries in the application menus
- The plugins can add their own widgets in a dedicated area of the application (typically a dock).
- A plugin can provide a map (see topological maps and CGoGN's map) that could be shared/reused by other plugins.

2.1.2 Additional features

In addition to the previous specifications it was also appreciate that the project could be completed with some features that are:

- A path for the cameras in a 3D scene could be interpolated from successive camera positions so that the camera could follow that path while she saves snapshots during its movement.
- A map could be imported from external files.

Those additional features, were developed as plugins for the application since they are not primary needed features.

2.1.3 Third party libraries

In order to implement the required features of the project in an easy and effective way, the project naturally uses third party libraries. Without considering the many third party libraries already used in **CGoGN**, the work was done using mainly two libraries:

- **Qt**: the effective and popular framework, which, thus providing all the GUI features, provides the kernel for our project's plugin system using Qt's plugin framework.
- **libQGLviewer**: A library that is also based on Qt, but that provides interesting efficient visualization features especially regarding camera gesture.

2.2 Quick presentation

This section quickly presents the main application interface then the use of the *importMap* and the *cameraPath* plugin.

2.2.1 The main application

Here's the main application after a first launch.

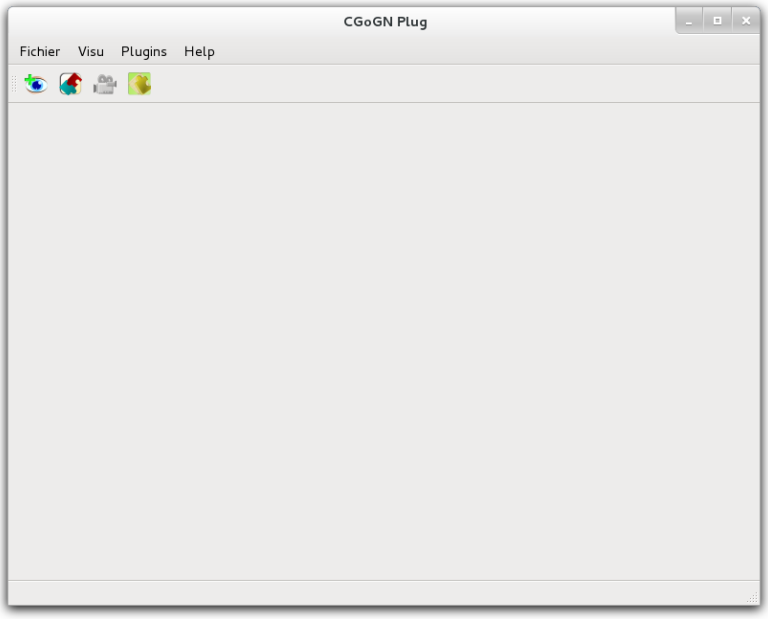


Figure 2.1: The main window application without any plugin loaded or used.

Now we need to load a plugin.

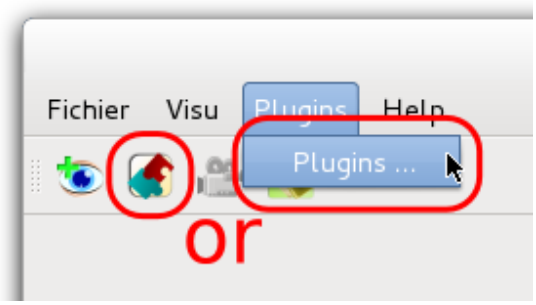


Figure 2.2: Load the plugin button.

We then choose a plugin to load. Plugins are particular C++ dynamic libraries, they take the form of *lib*.so* files. For example, here we want to load a plugin called *tuto5Geom* so we select he file *libTuto5Geom.so*. Custom plugin directories or, individual plugin files can be specified from this interface.

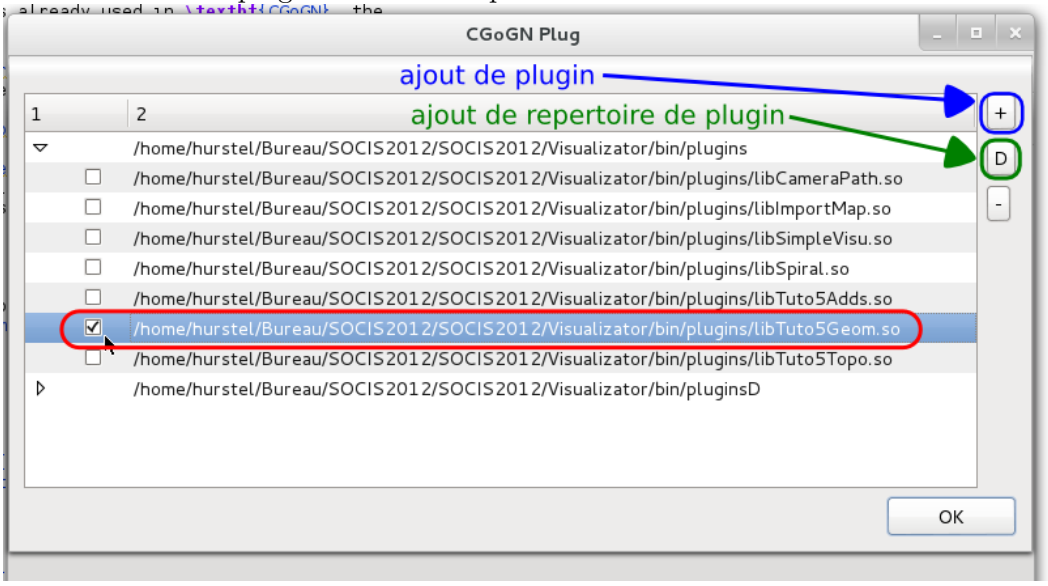


Figure 2.3: Load plugins interface.

Our plugin is loaded, but nothing happened. This plugin is a visualization plugin: it provides a visualization of a grid cube. We don't have yet any visualization because, most plugins generally wait to be associated with a scene to draw in, so we still have to create a scene manually. (Note that plugins could create their own scene automatically, if the plugin creator wants to, but this approach is discouraged)

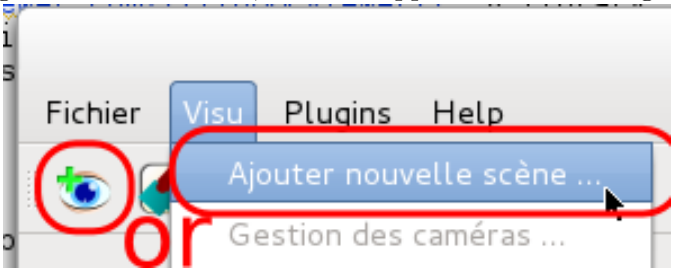


Figure 2.4: Adding scenes button.

We now chose a name for our view, and chose to link it with an existing previously load plugin. Note that a view can be linked to a plugin even after its creation.

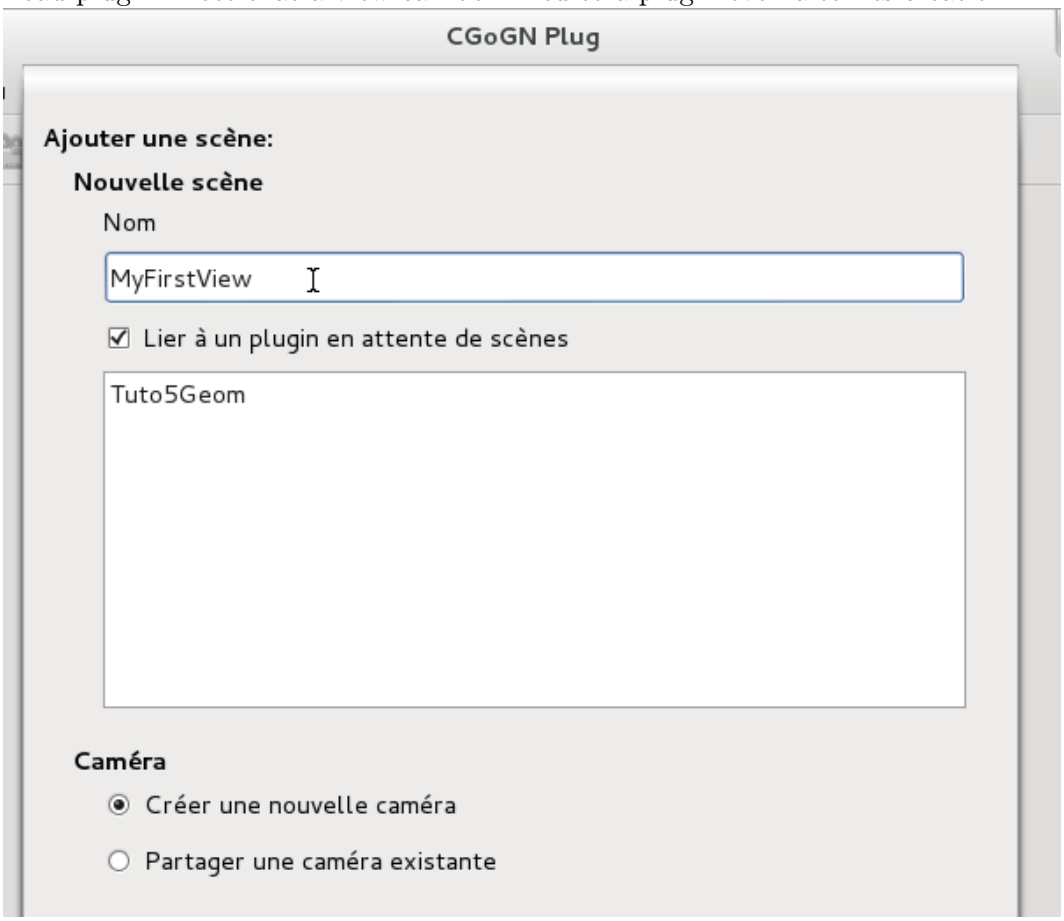


Figure 2.5: Adding scenes interface.

And here's the view with our drawing.

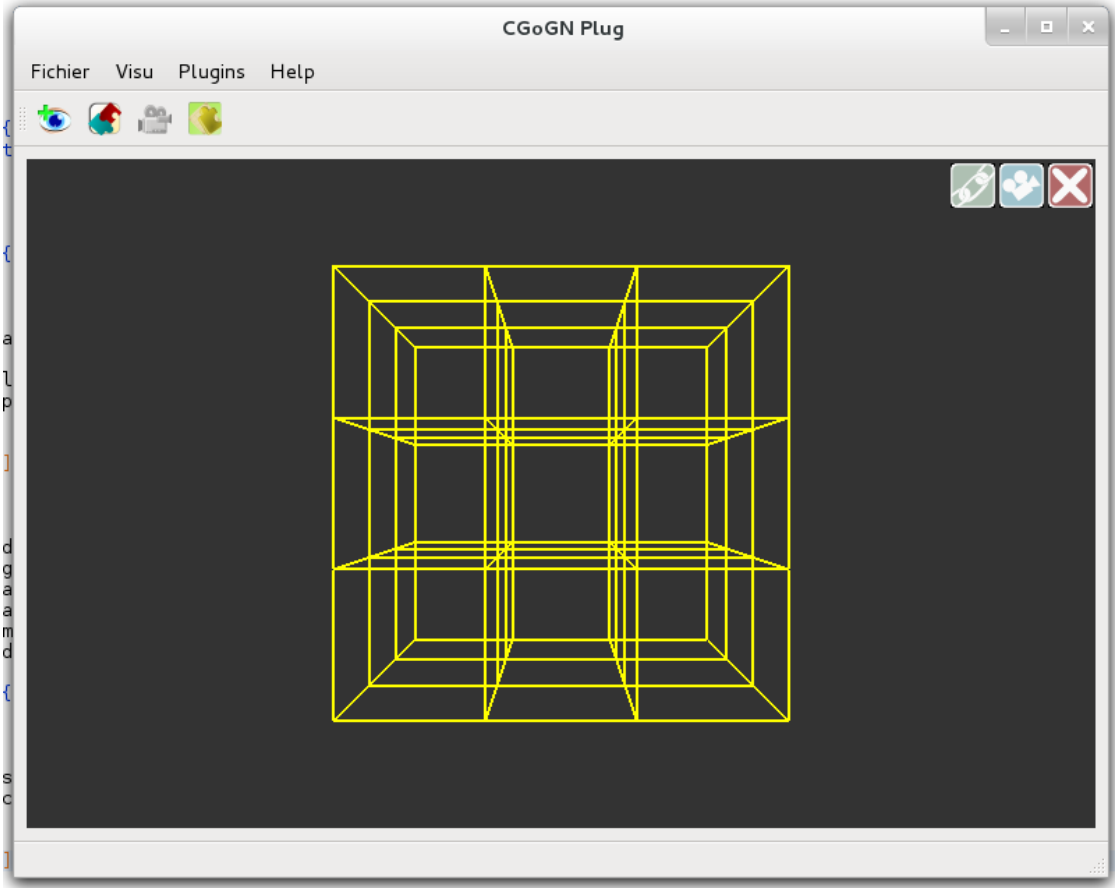


Figure 2.6: A scene associated to a drawing plugin.

Let's now watch this scene from two different points of view. Several step have to be made: first we have to create a new camera.

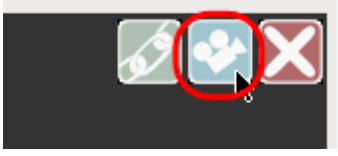


Figure 2.7: View's camera gesture for this scene.

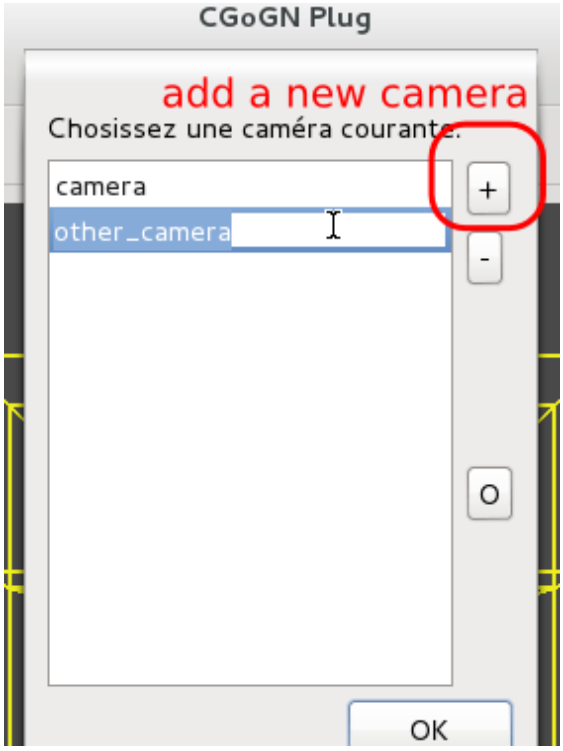


Figure 2.8: Creating and naming a new camera.

We now have a single view with two cameras. We're going to separate these two cameras into two different views, and by doing so splitting this scene into two views.

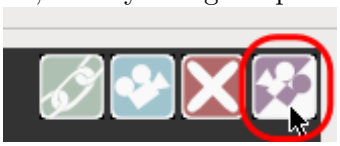
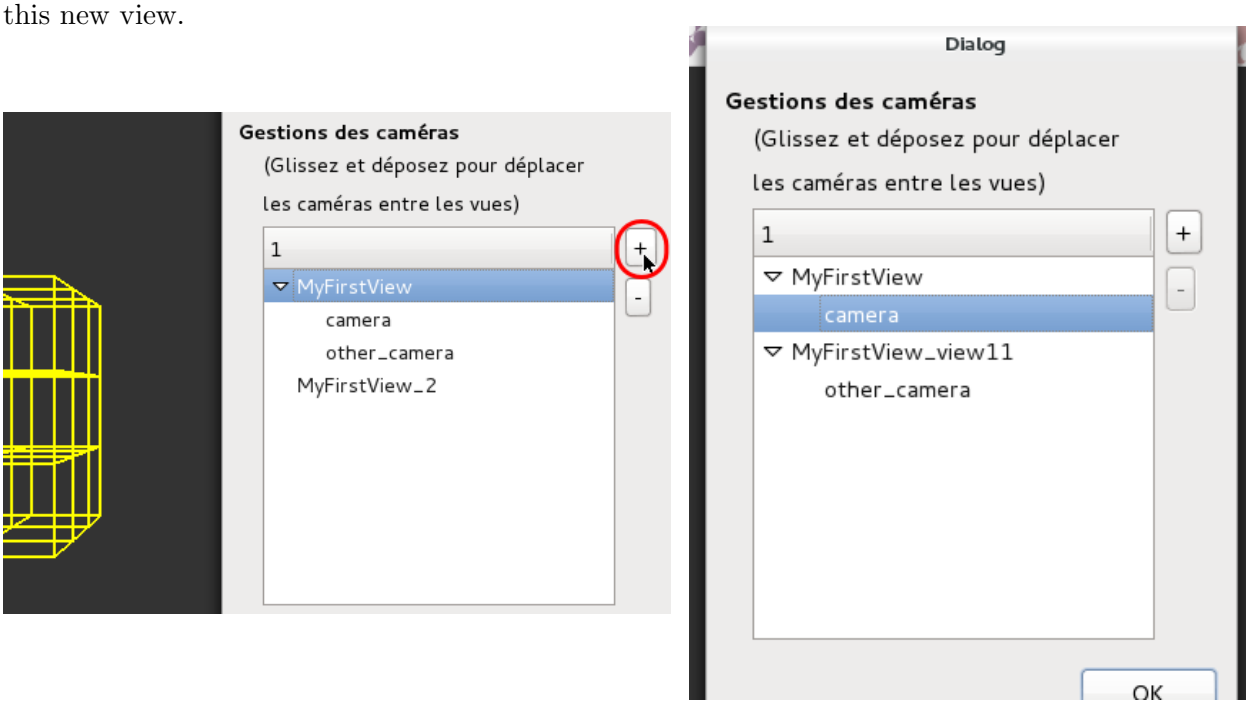


Figure 2.9: View management button.

We create a new view for this scene, then we drag'n drop one of the two cameras into this new view.



We now have the same scene that can be viewed by two different points of view.

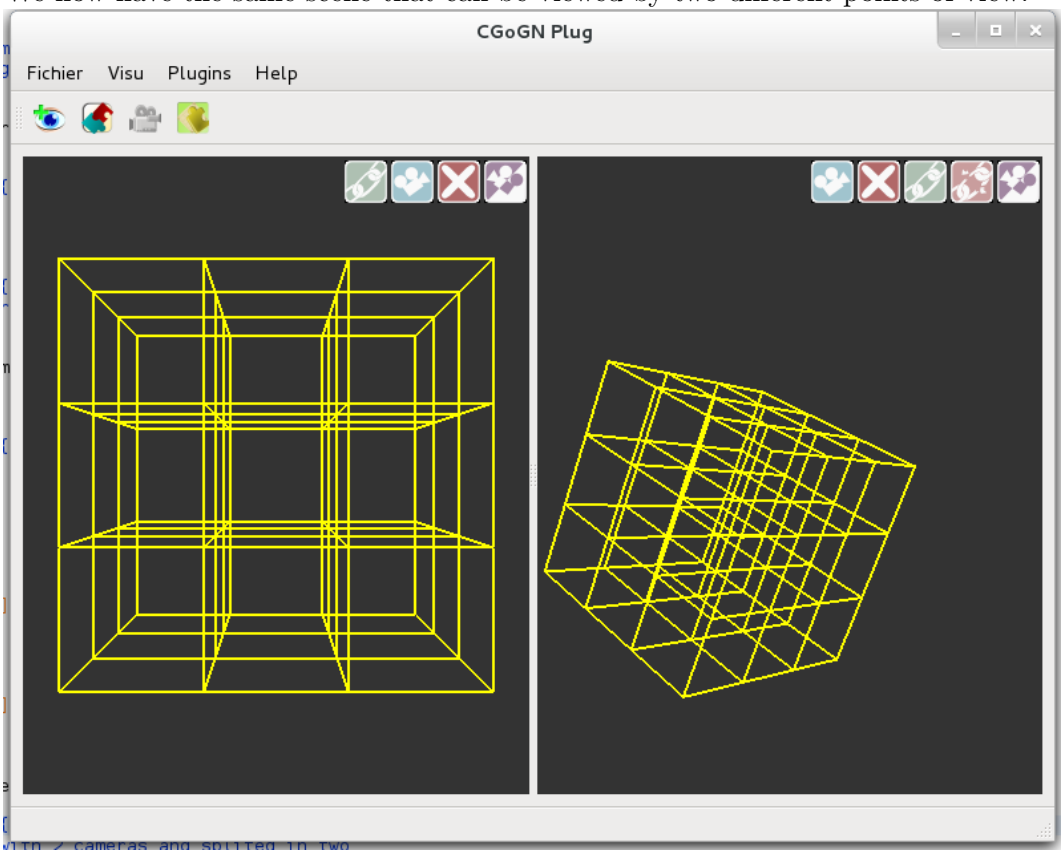


Figure 2.10: 1 plugin drawing in one scene with 2 cameras and split into 2 view.

2.2.2 The import plugin

The import plugin is a plugin that loads a map form a *.off* file, and then references it within the main application so that it can be given to other plugins and reused by them.

Loading the *importMap* plugin.

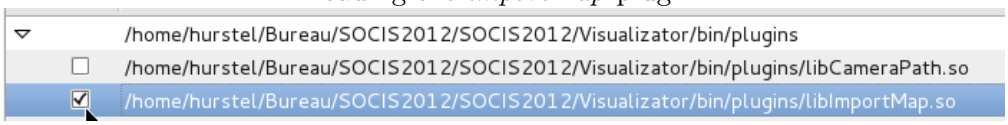


Figure 2.11: Loading the importMap plugin.

This plugin doesn't provide any visualization function, it just loads map into memory, that's why we don't (and can't) link it to any view. Once loaded the plugin adds an "*import map*" option to the menu and/or toolbar.

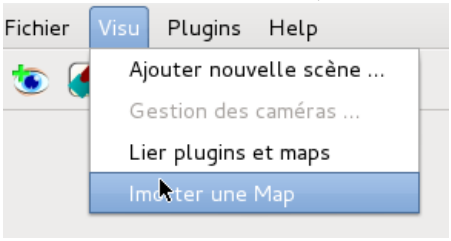


Figure 2.12: A wild menu entry appears!

We now chose an *.off* file to load.

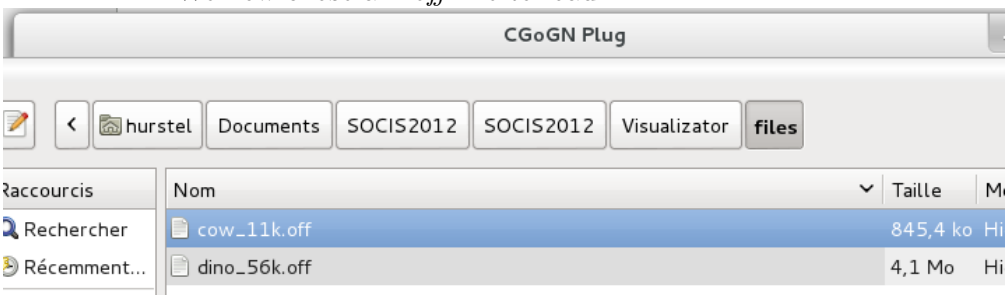


Figure 2.13: File dialog.

The map is now loaded in memory, but we need a tool to visualize it. Skipping the procedure that is the same as before, we load the "*simpleVisu*" plugin, and link a new scene to it. But we still have to tell the "*simpleVisu*" plugin, to use this map.

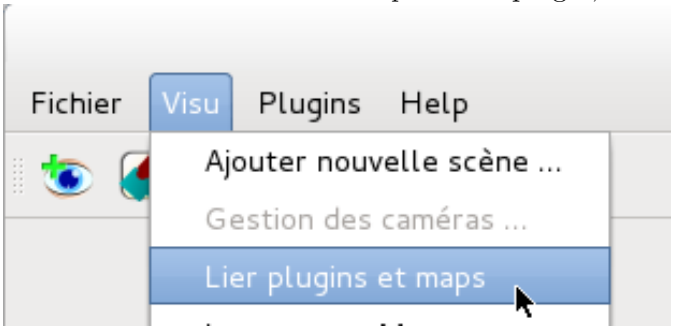


Figure 2.14: Link map and plugin menu entry

We select the concerned plugin, then we double click on the available maps list. The green icon indicates that the map is linked with the selected plugin.

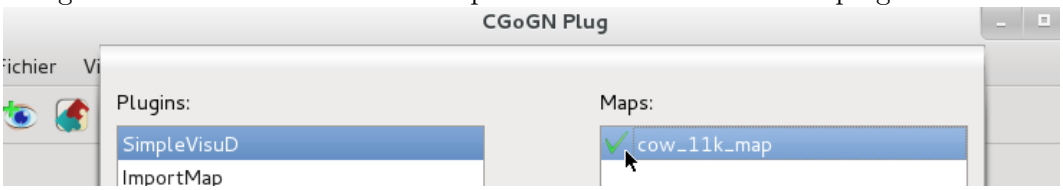


Figure 2.15: Link map and plugin menu entry

And we have a visualization of our loaded map.

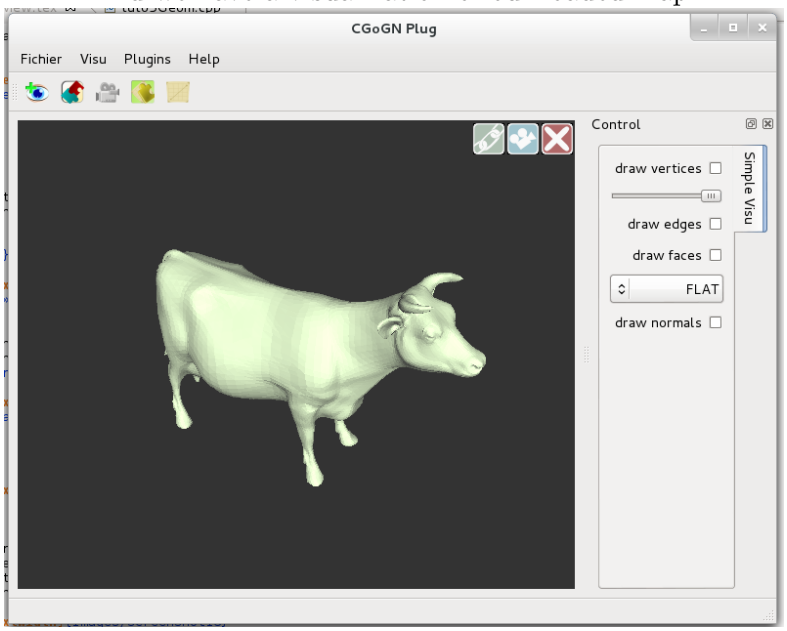


Figure 2.16: Moo moo mooo.

2.2.3 The camera path plugin

Another requirement for this project was to be able, given the multiple camera gesture, to interpolate a path for a camera from it's successive position defined by the user as "keyframes", so that the camera could move along this path and "replay" this movement on the user's demand. Additionally this path could be visible for other cameras.

We start with this double visualization of a same scene.

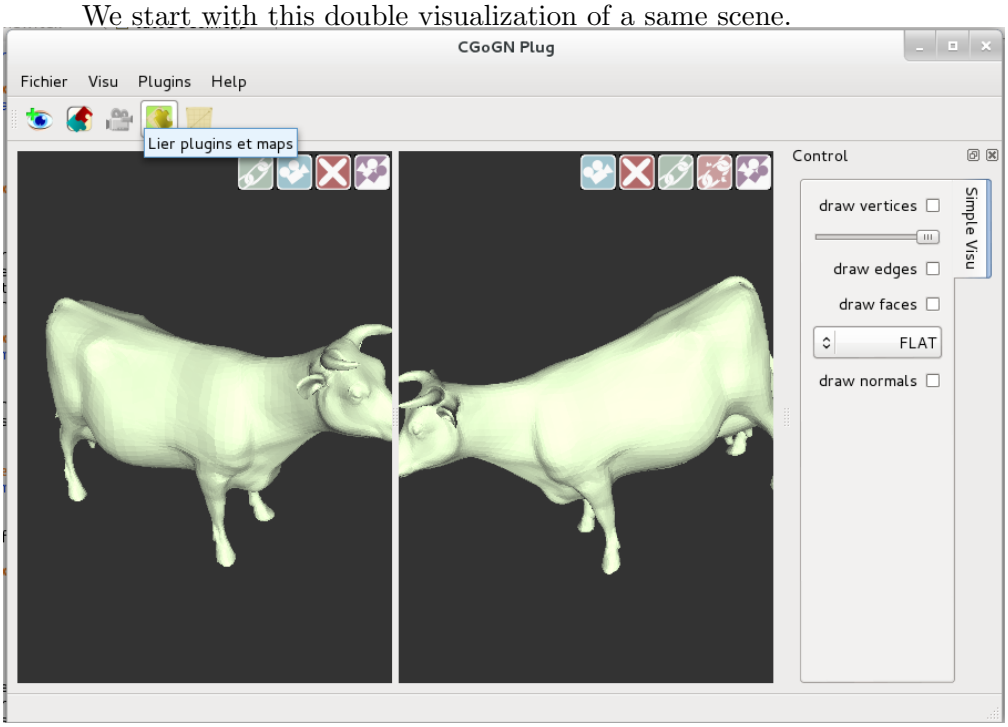


Figure 2.17: Two moos are better than none.

We now load the "*cameraPath*" plugin.

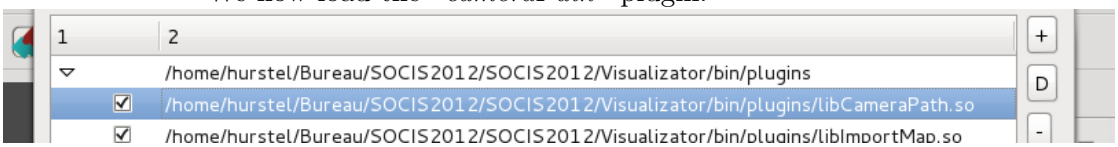


Figure 2.18: Regular plugin load.

Linking the existing scene with the plugin.

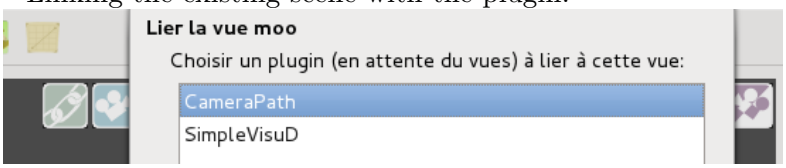


Figure 2.19: Linking the existing view with the "*cameraPath*" plugin.

A new icon appears on both views.

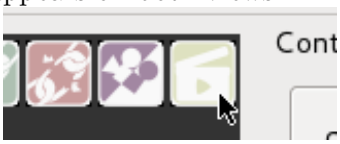


Figure 2.20: A wild new icon appears!

This opens a new dialog where a path can be created using successive positions of the camera. Each time the user hits the "create keyFrame" button, a new keyframe, which is the current position of the view's current camera, is added to the path. The path is interpolated from this keyframe list.

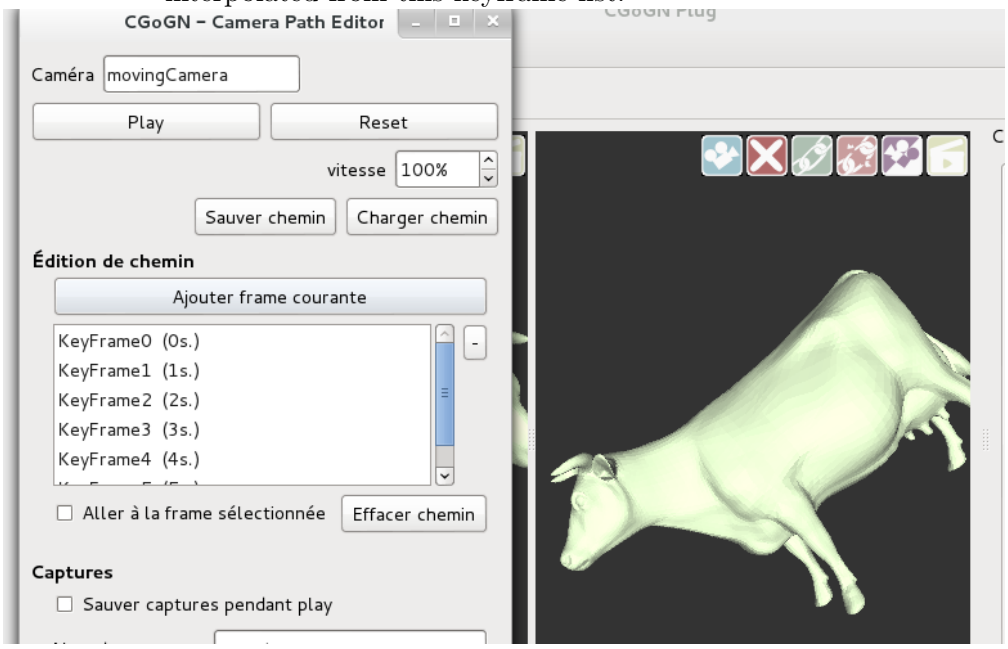


Figure 2.21: The camera path editor dialog.

The camera and its interpolated path can both be made visible by other cameras from the same scene.

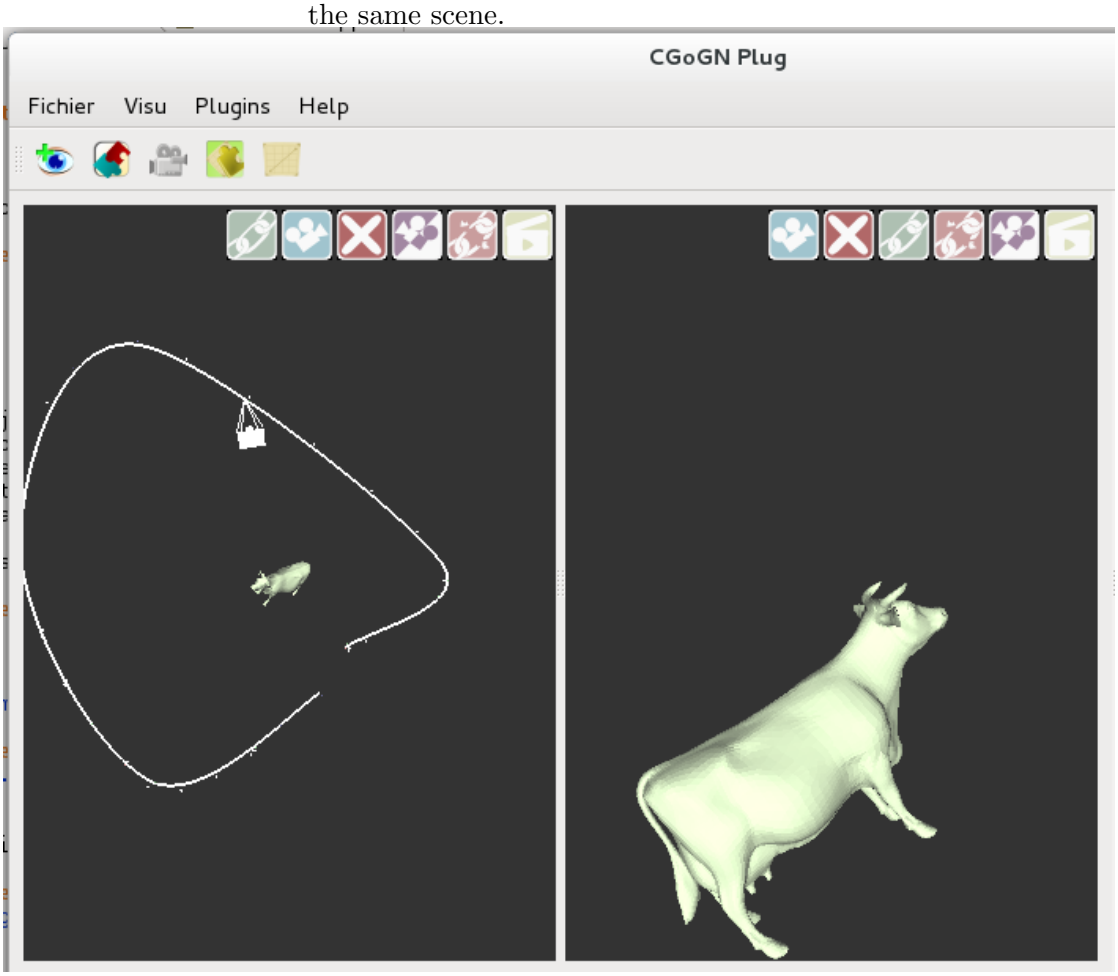


Figure 2.22: The path of the second camera viewed by the first.

A demo video showing how this plugin works may have been joined with this document, or may be found in the project's documentation.

3 How to write plugins?

3.1 Basics and concepts

This sections intends to explain the basis to write a visualization plugin. It assumes you have knowledge about compilation of C++ **Qt** files, and also uses **CMake**, a powerful Makefile generating tool. The goal of this section is to write a plugin version of this simple **CGoGN** example: `http://cgogn.u-strasbg.fr/Wiki/index.php/Tuto_1`

3.1.1 Visualization concepts

Before writing visual plugins there are some main application inner mechanisms that need to be understood, that is to say, the way the main application manages the different objects of interest and their relations between each other: **Camera**, **View**, **Scene** and **Plugin**.

Plugin: A plugin is a piece of code written independently to the main application, that are loaded and referenced from the main application, and that are instantiated as a new object: **Plugin**. If a plugin has an active drawing method, it can draw any visualization but it needs to be linked to another object that exists within the main application: a **Scene**. Note that a plugin does not need to be linked to a scene, if it doesn't provide any drawing.

Scene: It's an object that is created by the main application, generally on user's demand. A scene isn't actually a drawing object, its aim is to make the link between two types of objects: the **View** and the **Plugin**. A scene can be defined as a set of plugins. A scene exists only for the visualization of the plugin drawing methods, so it has to contain at least one **View**. A scene can be linked to several plugins, and a plugin can be linked to several scenes.

View: It's the visualization object that renders the drawings of the plugin that are associated to it's parent's **Scene**. Whereas a scene can have several views, a view only have one parent scene. The views can switch between several point of views, using **Camera** objects. A view has at least one camera.

Camera: This objects can be seen as a point of view. A **View** has a least one camera, and has necessarily an unique "current" camera, that is it's current point of view. A **View** can switch between several camera, and it also can be considered as a collection

of cameras. A **Camera** can belong to several views (from different scenes) at the same time, it is then a "shared" camera.

Here is a simple representation of a simple situation: a drawing **Plugin** that is linked to a **Scene** that is made of an unique **View** containing only one **Camera** (ie: 2.6):

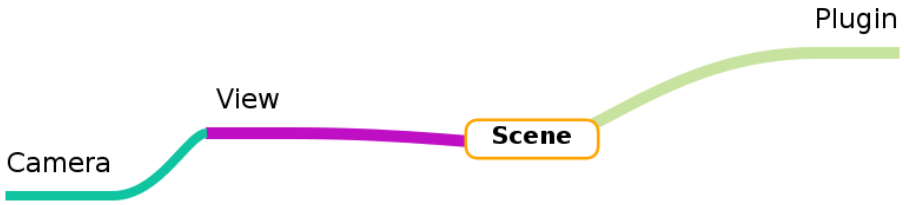


Figure 3.1: 1 plugin drawing in 1 scene made of 1 view that has 1 camera.

Furthermore, a representation of (2.10) would be:

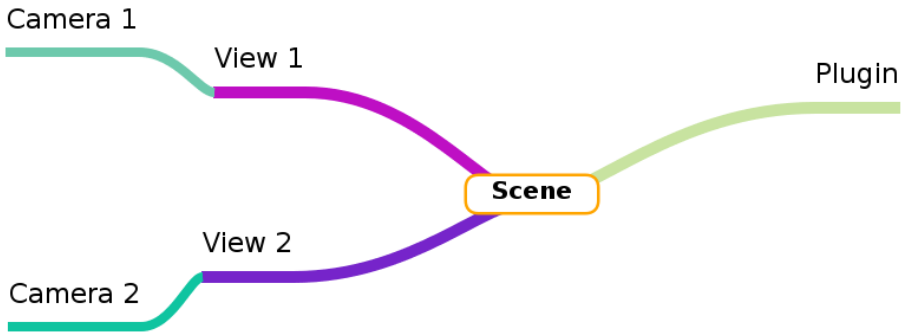


Figure 3.2: 1 plugin drawing in 1 scene made of 2 views each one working with their own camera.

3.1.2 The first plugin

How to write the plugin

A good way to comprehend the plugin mechanism is to check the **Qt** plugin mechanism, which our plugin system relies on.

Our plugin is based on a **CGoGN** tutorial: `http://cgogn.u-strasbg.fr/Wiki/index.php/Tuto_1`. Here, we will create a single visualization plugin that creates in a map a triangle, a square, connect them, affect positions to vertices and visualize it. The

plugin will be called "*FirstPlugin*", and will be compiled as a dynamic C++ library, that will consequently result in a file called "**libFirstPlugin.so**". Here's the skeleton for such a plugin:

firstPlugin.h

```
1   #ifndef FIRSTPLUGIN_H_
2   #define FIRSTPLUGIN_H_
3
4
5
6   #include "visualPlugin.h"
7
8
9   /**---CGoGN includes **/
10    #include "Utils/Qt/qtSimple.h"
11    #include "Utils/cgognStream.h"
12
13    #include "Topology/generic/parameters.h"
14
15    #ifdef USE_GMAP
16      #include "Topology/gmap/embeddedGMap2.h"
17    #else
18      #include "Topology/map/embeddedMap2.h"
19    #endif
20
21    #include "Algo/Render/GL2/topoRender.h"
22  /**---CGoGN includes **/
23
24
25  /**---Definitions  specific to CGoGN ---*/
26    using namespace CGoGN ;
27
28    /**
29     * Struct that contains some information about the types of ↩
          the manipulated objects
30     * Mainly here to be used by the algorithms that are ↩
          parameterized by it
31     */
32    struct PFP: public PFP_STANDARD
33    {
34      // definition of the type of the map
35    #ifdef USE_GMAP
36      typedef EmbeddedGMap2 MAP;
37    #else
38      typedef EmbeddedMap2 MAP;
39    #endif
40    };
```

```
41
42        typedef PFP::MAP MAP;
43        typedef PFP::VEC3 VEC3;
44      /**---Definitions  specific to CGoGN ---*/
45
46      /**
47       * This class is a basic minimal plugin.
48       * All the methods in this class are overloaded methods.
49       * In order to create a valid plugin, all the method in this
50       * needs to be declared (they are actually overloaded methods
51       * from VisualPlugin), even if your plugin doesn't make any
52       * drawing.
53       */
54
55      /**
56       * Our plugin must inherit from VisualPlugin,
57       * that is a class that itself is an implementation
58       * of the Plugin interface (virtual class). It contains
59       * many useful and essential methods.
60       */
61      class FirstPlugin : public VisualPlugin{
62        /**
63         * Essential Qt macros.
64         */
65        Q_OBJECT
66        Q_INTERFACES(Plugin)
67      public:
68        FirstPlugin(){}
69        ~FirstPlugin(){}
70
71        /**
72         * The classical call back for the initGL method
73         * When a scene will be link to this plugin, it will call
74         * back this method with itself as a parameter.
75         */
76        void cb_initGL(Scene* scene);
77
78        /**
79         * The drawing method that needs to be overloaded.
80         * Each time a scene (that is to say, at least one of the
81         * views that is contains) needs to be refresh, it calls back
82         * this method with itself as a parameter
83         */
84        void cb_redraw(Scene* scene);
85
86        /**
87         * The plugin's activation method
88         * Each time the main application loads this plugin,
89         * it call this method. Writing this method is
```

```
90          * the occasion to initialize the plugin and check certain
91          * conditions.
92          * If this methods return 'false', the plugin load will be ↩
                aborted.
93          */
94         bool activate();
95         /**
96          * The plugin's disabling method
97          * Each time the main application will unload the plugin
98          * it will call this method.
99          */
100        void disable();
101
102
103     protected:
104       /** Attributes that are specific to this plugin **/
105         MAP myMap;
106
107         // attribute for vertices positions
108         VertexAttribute<VEC3> position;
109
110         // render (for the topo)
111         Algo::Render::GL2::TopoRender* m_render_topo;
112
113         // just for more compact writing
114         inline Dart PHI1(Dart d) { return myMap.phi1(d); }
115         inline Dart PHI_1(Dart d) { return myMap.phi_1(d); }
116         inline Dart PHI2(Dart d) { return myMap.phi2(d); }
117         template<int X>
118         Dart PHI(Dart d) { return myMap.phi<X>(d); }
119       /** Attributes that are specific to this plugin **/
120    };
121
122    #endif /* FIRSTPLUGIN_H_ */
```

And here is the *.cpp* file, that implements these methods:

firstPlugin.cpp

```
1    #include "firstPlugin.h"
2
3    #include "Algo/Geometry/boundingbox.h"
4
5
6    bool FirstPlugin::activate(){
7      // creation of 2 new faces: 1 triangle and 1 square
8        Dart d1 = myMap.newFace(3);
```

```
9           Dart d2 = myMap.newFace(4);
10
11          // sew these faces along one of their edge
12          myMap.sewFaces(d1, d2);
13
14          // creation of a new attribute on vertices of type 3D ↩
                vector for position.
15          // a handler to this attribute is returned
16          position = myMap.addAttribute<VEC3, VERTEX>("position");
17
18          // affect position by moving in the map
19          position[d1] = VEC3(0, 0, 0);
20          position[PHI1(d1)] = VEC3(2, 0, 0);
21          position[PHI_1(d1)] = VEC3(1, 2, 0);
22          position[PHI<11>(d2)] = VEC3(0, -2, 0);
23          position[PHI_1(d2)] = VEC3(2, -2, 0);
24
25          m_render_topo=NULL;
26
27          return true;
28      }
29
30      void FirstPlugin::disable(){
31        if(m_render_topo){
32          delete m_render_topo;
33        }
34      }
35
36      void FirstPlugin::cb_redraw(Scene* scene){
37        m_render_topo->drawTopo();
38      }
39
40      void FirstPlugin::cb_initGL(Scene* scene){
41        if(scene){
42          //we fit the first (possibly the only) view
43          //of the newly linked scene to the content
44          //of our map
45
46          //  bounding box of scene
47          Geom::BoundingBox<PFP::VEC3> bb = Algo::Geometry::↩
                computeBoundingBox<PFP>(myMap, position);
48
49          scene->firstViewFitSphere(bb.center()[0], bb.center()[1], ↩
                bb.center()[2], bb.maxSize());
50
51
52
53          m_render_topo = new Algo::Render::GL2::TopoRender() ;
54
```

```
55          // render the topo of the map without boundary darts
56          SelectorDartNoBoundary<PFP::MAP> nb(myMap);
57          m_render_topo->updateData<PFP>(myMap, position, 0.9f, 0.9f,↩
                nb);
58
59        }
60      }
61
62
63      /**
64       * If we want to compile this plugin in debug mode,
65       * we also define a DEBUG macro at the compilation
66       */
67      #ifndef DEBUG
68        //essential Qt function:
69        //arguments are
70        //  - the complied name of the plugin
71        //  - the main class of our plugin (that extends VisualPlugin↩
              )
72        Q_EXPORT_PLUGIN2(FirstPlugin, FirstPlugin)
73      #else
74        Q_EXPORT_PLUGIN2(FirstPluginD, FirstPlugin)
75      #endif
```

Compilation

To compile a plugin, the files must be compiled into a C++ dynamic library with the plugin's main class compiled as a **Qt** object using the **Qt** tools.

Assuming that you are already familiar with **CMake** and compiling **CGoGN** apps, here's a cmake example, compiling the plugin written above:

CMakeLists.txt

```
1       #assuming that ${QGLVIEWER_PATH} is the directory
2       #where QGLViewer was compiled
3       SET(QGLVIEWER_INCLUDE_DIR ${QGLVIEWER_PATH}/QGLViewer)
4
5       #the include directories for CGoGN, common libraries
6       #and QGLViewer
7       INCLUDE_DIRECTORIES(
8         ${CGoGN_ROOT_DIR}/include
9         ${COMMON_INCLUDES}
10        ${CMAKE_CURRENT_SOURCE_DIR}
11        ${CMAKE_CURRENT_BINARY_DIR}
12        ${QGLVIEWER_INCLUDE_DIR}
13      )
```

```
14
15      #assuming that ${CGoGN_PLUG_DIR} is the directory
16      #of main application project was compiled
17      SET(VISUALPLUGIN_CPP_PATH
18        ${CGoGN_PLUG_DIR}/plugin/visualPlugin.cpp
19      )
20      SET(VISUALPLUGIN_H_PATH
21        ${CGoGN_PLUG_DIR}/plugin/visualPlugin.h
22      )
23
24      SET( FIRSTPLUGIN_SRC
25        ${VISUALPLUGIN_CPP_PATH}
26          firstPlugin.cpp
27      )
28      SET( FIRSTPLUGIN_H
29        ${VISUALPLUGIN_H_PATH}
30          firstPlugin.h
31      )
32
33      #using Qt tools to generate the "moc" file for the
34      #plugin Q_OBJECT
35      QT4_WRAP_CPP(FIRSTPLUGIN_MOC ${FIRSTPLUGIN_H})
36
37      #compiling as a dynamic library
38      ADD_LIBRARY(FirstPlugin SHARED
39        ${FIRSTPLUGIN_SRC}
40        ${FIRSTPLUGIN_MOC}
41      )
42
43      #linking with the CGoGN and QGLViewers libraries
44      TARGET_LINK_LIBRARIES(FirstPlugin
45        ${CGoGN_LIBS_R}
46        ${COMMON_LIBS}
47        ${QGLVIEWER_INCLUDE_LIBRARY}
48      )
```

The compilation should generate our plugin file: "*libFirstPlugin.so*".

Using the plugin

Before using our plugin, we must add it to the main application's plugin location list. First we have to open the "plugin interface".

On the left the two first button are respectively to add and individual plugin's location, and to add a plugin directory location so that all the "*lib*.so*" files in this directory will be listed.

Figure 3.3: Add a plugin and add a plugin directory buttons.

We choose the first option, and our load plugin individually.

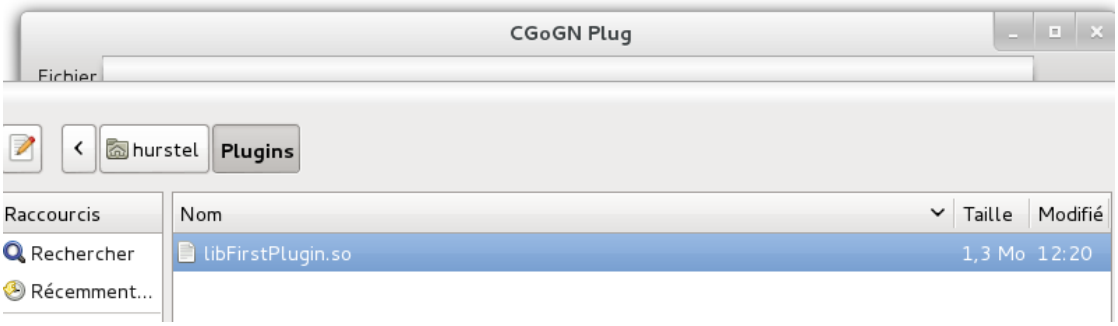


Figure 3.4: We select our plugin.

We can now load our plugin and link it to a view.

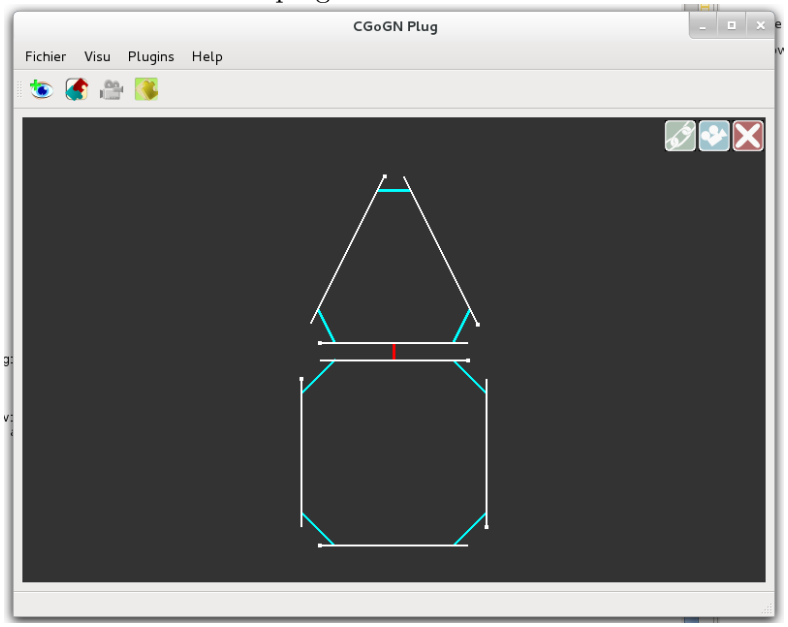


Figure 3.5: Tadaaaa!

3.1.3 Tricks and advice

Understanding the callBacks

To avoid segmentation faults it may be of a good help to understand how the plugin's call back methods are called. The figure (3.6) should be of an help.

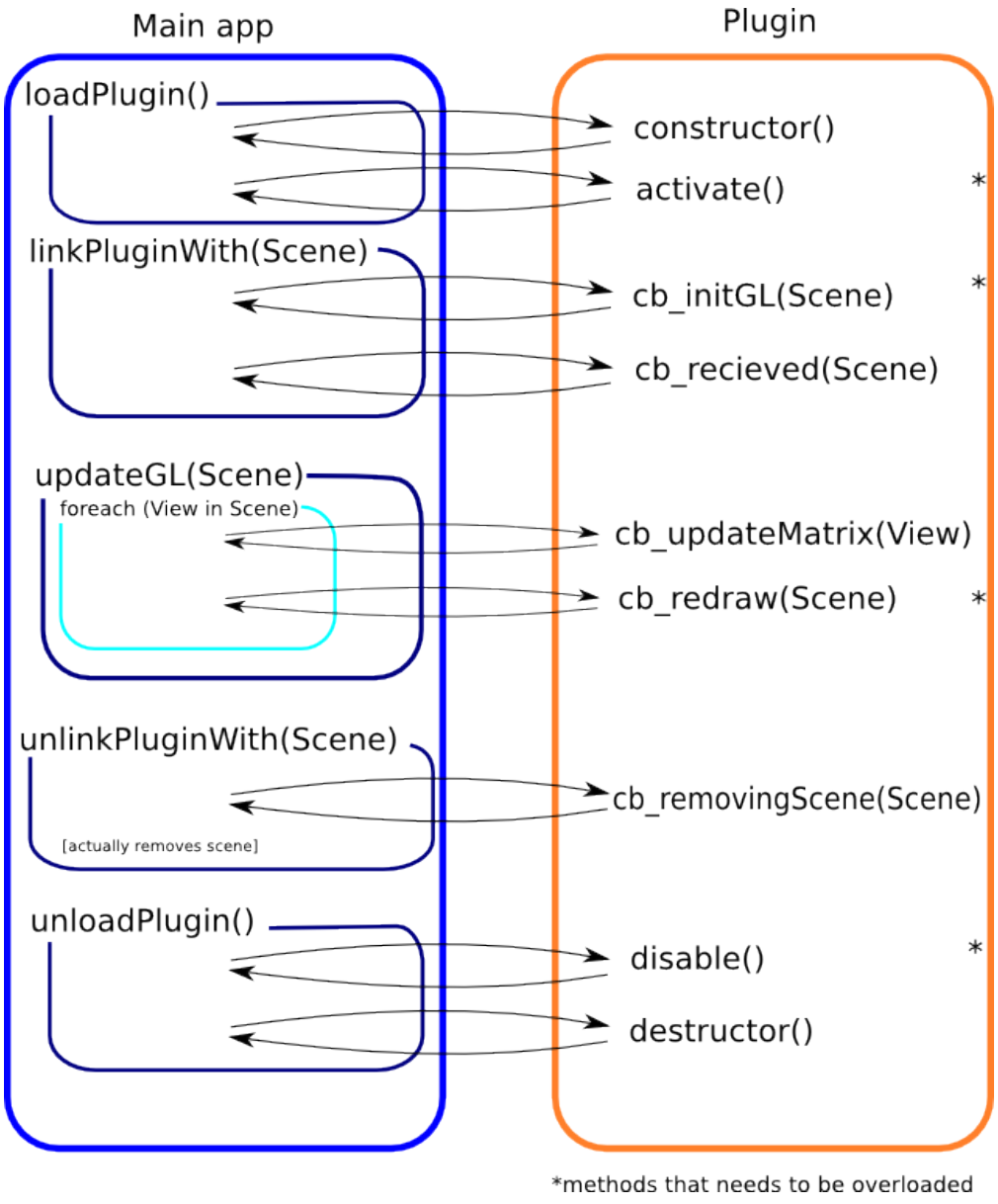


Figure 3.6: The main call backs between the main application and a plugin.

Bug & Debug

Common bug: As said before, the compiled are "*lib*.so*" files. But sometimes, when you try to load one of these files as a plugin, the program may tell you that the plugin isn't compatible. Several simple mistakes can be the cause of this inconvenience. Here's some advice to avoid it:

- make sure that all the necessary Qt macros are correctly used:
  - in the plugin's main class declaration: `Q_OBJECT` and `Q_INTERFACES(Plugin)`
  - in the plugin's implementation file: `Q_EXPORT_PLUGIN2(ProgName, MainClassName)`
- make sure that you have overloaded each of the four obligatory **VisualPlugin** pure virtual methods:
  - `bool activate()`
  - `void cb_initGL()`
  - `void cb_redraw()`
  - `void disable()`
- make sure that the files containing the declaration and implementation of the plugin's main class are compiled as **Qt** object files.
- make sure that each method you declare or overload is implemented.
- *do not* redefine (change signature) any of the **VisualPlugin**'s inherited method.

Debug: Naturally a coder will want to test and debug its plugin. If you want to use **GDB** on your own plugin their are a few steps you need to follow:

- Compile your plugin in debug mode. Be sure that you use the **debug** version of the CGoGN library. If you have a normal built plugin and a debug built plugin with different names, in your plugin code, be sure to use:

```
#ifndef DEBUG
  Q_EXPORT_PLUGIN2(FirstPlugin, FirstPlugin)
#else
  Q_EXPORT_PLUGIN2(FirstPluginD, FirstPlugin)
#endif
```

  and to define the `DEBUG` macro in your debug compilation.

- Use a debug compiled version of the main application with **GDB**
- In **GDB** before using the `run` command don't forget to use:

```
directory [path_to/the_directory/containing/the_plugin]
```

If you follow these steps you should be able to debug your plugin using **GDB**.

3.2 Going further...

This section aims to present and explain more plugin features and methods in order to write more advanced plugins.

3.2.1 Few words on plugins

As seen in the previous plugin example, a plugin must be an inheritance of a virtual class, **VisualPlugin**, that is itself an inheritance of a pure virtual class, considered as a plugin interface (3.7).

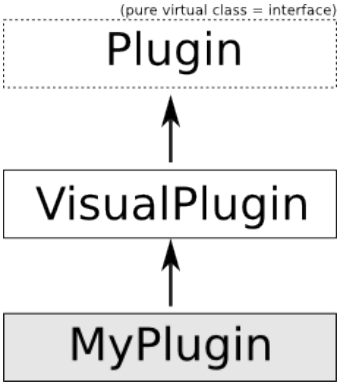


Figure 3.7: The plugins inheritence graph.

The point of inheriting an halfway class is that **VisualPlugin** already declares some empty virtual call-back methods so that the plugin coder doesn't have to declare the ones he doesn't want to use, but it also provides several *ready-to-use* methods that offer many easy to use features, for example, add GUI interface, adjust some plugins properties. Many of these methods are presented and explained further.

3.2.2 Objects and visualization plugins

As you know, one of the main goals of this project is to provide a main application that is a mere kernel that needs to be completed with third party plugins. The main application provides several objects for visualization and interface creation. As told earlier, this application and the plugin systems uses **Qt** and another library (also Qt based) **libQGLviewer**. Here are some precision about the objects of interest that you may have to consider for plugin writing.

Scene:

A scene is a drawing that can be made by one or several plugin at the same time. A scene can be scene from several point of views (View objects) at the same time. That is why a scene is both a collection of plugins and a collection of views.

Automatic creation: As seen earlier, a scene is meant to be created empty (with one view, and linked to no plugin) by user, then be linked by the user to an active drawing plugin. Although a plugin can automatically create a scene if he calls the method:

```
bool addNewScene(QString name, Scene* &scene)
```
or
```
bool addNewSceneDialog(QString name, Scene* &scene)
```

The only difference between two methods is that the first automatically places the first view of the scene, and the second one opens a dialog to ask the user where to place it on the "view area" of the application. The effects and signatures for these methods use are the same:

	name	type	description
argument:	name	QString	The name under witch the scene will be referenced in the main application.
	scene	Scene* &	Once the scene is created, this pointer will be affected with it's memory address: it's an out parameter.
return:		bool	A boolean whether the creation of the scene has succeeded or not.

Notes:

- A call to the plugin's `cb_intGL()` call back method will be made but not to `cb_recieveScene()`.
- When the plugin will be unload, this scene will be destroyed.
- Creating a scene automatically *is not recommended.*

Call-back on scene linking: We've mentioned several time the call-back method:

```
void cb_recievedScene(Scene* scene)
```

As you may have guessed, it's a call-back method that is called each time a scene is linked to the plugin by the user.

	name	type	description
argument:	scene	Scene*	A pointer on the scene that was linked by the user to the plugin.

The plugin maintains a list attribute `l_recievedScene` that contains all the references to the received scene. The `cb_recievedScene` method is called *after* the scene has been added to this list. You do not have to, and should not, operate on this list (the only operations that could be allowed are sorting or re-ordering the list). Overload this method each time you want a specific action to be done, when a scene is linked to your plugin.

Of course their's also a call-back method that is called when the scene is unlinked and that works in a similar way:

```
void cb_removingScene(Scene* scene)
```

However, be sure to note that this method is called *before* the scene is removed from the `l_recievedScene` list. Once again you do not have to, and should not, operate on this list. Overload this method each time you want a specific action to be done, when a scene is unlinked from your plugin.

View:

The **View** object is the object that renders the drawing (drawn by one or more plugins) that characterize a scene. A view can be considered as a point of view, and so a scene can have multiple view, although a view belongs to a one and only scene. The view inherits the **QGLViewer** class (from the **libQGLviewer** library). So if you want to operate directly on a view, be sure to check the library's documentation.

You can access to a scene's views using some of **Scene**'s public methods:

`int countViews()`

	name	type	description
return:		int	The current number of views of a scene.

`View* getView(int num)`

	name	type	description
argument:	num	int	the scene's *num*-th view you want to access.
return:		View*	A pointer to the *num*-th view of the plugin, NULL if the view can't be accessed or doesn't exists.

`QList<View*> views()`

	name	type	description
return:		QList<View*>	A list containing references to all of the scene's views.

Accessing and operating directly on a scene's view can be critical: be very cautious. Although this might be useful in some cases, especially when you want to access the current camera's matrices, which is common when you overload the callback method `cb_updateMatrix(View*)`; if you are already familiar with **CGoGN**'s shaders implementation, the default implementation of the method may enlighten you:

```
1   void VisualPlugin::cb_updateMatrix(View* view){
2     if(view){
3       glm::mat4 model(view->getCurrentModelViewMatrice());
4       glm::mat4 proj(view->getCurrentProjectionMatrice());
5
6       for(std::set< std::pair<void*, Utils::GLSLShader*> >::↩
            iterator it = Utils::GLSLShader::m_registeredShaders.↩
            begin();
7         it != Utils::GLSLShader::m_registeredShaders.end();
8         ++it)
9       {
10        if ((it->first == NULL) || (it->first == this))
11        {
12          it->second->updateMatrices(proj, model);
13        }
14      }
15    }
16  }
```

Camera:

As explained earlier the **View** object has a collection of *Camera* objects. These cameras can be considered as different point of views available for a view, of course only one camera is considered current for a given view. A camera can be shared by several views, so it's modification moreover it's suppression can be critical. Several public methods of the **View** class can be called.

`Camera* currentCamera()`

	name	type	description
return:		Camera*	A pointer to the view's current camera.

`QList<Camera*> cameras()`

	name	type	description
return:		QList<Camera*>	A list containing references to all the cameras of the view.

`int countCameras()`

	name	type	description
return:		int	The number of cameras for the view.

The class **Camera** inherits the class **qglviewer::Camera** provided by the **libQGLViewer** library. If you want to operate correctly on cameras you might want to check it's documentation, but again, it also might be critical.

3.2.3 GUI and user interactions

Custom widgets and menu entries

Of course the plugins also offers the possibility to add you own menu entries, buttons in the toolbar and custom widget in the main application's dock. Although these features requires the basics of **Qt**'s custom slots and custom widget creation.

Custom menu entries: Their are two usable methods provided by the inheritance to **VisualPlugin** that allows you to add your custom menu entries:

```
QAction* addMenuAction(QString menuPath, const char* method)
```

	name	type	description
argument:	menuPath	QString	The "menu path" for your custom entry matching this pattern: *:menu/submenu1/submenu2/myEntry*. For example if your menu path is *:file/export/image*, in will create a submenu "export", in the "file" menu, that contain the entry image. Any non existing menu or submenu is created by this method.
	method	const char*	The **Qt** slot you want to connect to your menu entry. For example if you want the slot "`exportImage()`" to be called when the custom entry is clicked, this argument should be "SLOT(exportImage())".
return:		QAction*	A reference to the Qt's "action" object that has been created by this method. NULL if the method fails.

This method creates automatically your "action" object that is used to connect your entry with one of your slots. Although you might want to connect a same action to several menu entries or a to a toolbar button. In that case you you can declare your own QAction instance, and use this method:

```
bool addMenuAction(QString menuPath, QAction* action)
```

	name	type	description
argument:	menuPath	QString	The "menu path" for your custom entry matching this pattern: *:menu/submenu1/submenu2/myEntry*. For exemple if your menu path is *:file/export/image*, in will create a submenu "export", in the "file" menu, that contain the entry image. Any non existing menu or submenu is created by this method.
	method	QAction*	A pointer on the QAction you want to connect to your custom entry.
return:		bool	True if the method succeeded, false otherwise.

Custom toolbar buttons: In a similar way **VisualPlugin** provides inherited methods that can be used to add custom toolbar buttons:

```
QAction* addToolbarAction(const char* method, QIcon icon= QIcon())
```

	name	type	description
argument:	method	const char*	The **Qt** slot you want to connect to your toolbar button. For example if you want the slot "`exportImage()`" to be called when the custom entry is clicked, this argument should be "SLOT(exportImage())"
	icon	QIcon	The **Qt**'s icon of your custom button, an empty icon by default.
return:		QAction*	A reference to the Qt's "action" object that has been created by this method. NULL if the method fails.

In the same way than before, there's also a method where you passes an already created QAction instance:

```
bool addToolbarAction(QAction* action)
```

	name	type	description
argument:	action	QAction*	A pointer on the QAction you want to connect to your custom entry.
return:		bool	True if the method succeeded, false otherwise.

This last method considers that their's already a QIcon object associated with you the QAction object you passed as parameter.

Custom widgets: The following method supposes that you are already familiar with how to create (and compile) custom **Qt** widgets. The main application possesses a dock where widgets can be add as new tabs in the dock's area. This method allows you to add such widgets:

`bool addWidgetInDockTab(QWidget* newTabWidget, QString tabText)`

	name	type	description
argument:	newTabWidget	QWidget*	A reference to the custom widget you want to add as a new tab in the main application's dock.
	tabText	QString	The text that appears on the tab that contains your custom widget.
return:		bool	True if the method succeeded, false otherwise.

User interactions

Of course a plugin coder will want to control the behavior when mouse or keyboard actions are detected on a view of a scene. Naturally the **VisualPlugin** class that your plugin must inherit provides callback methods that you can overload. It is believe that these methods names are explicit enough so that they don't need to be described:

- `bool cb_keyPress(Scene* scene, int event)`
- `bool cb_keyRelease(Scene* scene, int event)`
- `bool cb_mousePress(Scene* scene, int button, int x, int y)`
- `bool cb_mouseRelease(Scene* scene, int button, int x, int y)`
- `bool cb_mouseClick(Scene* scene, int button, int x, int y)`
- `bool cb_mouseMove(Scene* scene, int buttons, int x, int y)`
- `bool cb_wheelEvent(Scene* scene, int delta, int x, int y)`

You will notice that all of these callback methods must return a boolean. If they return false, then the default behavior of a **QGLViewer** object (**View** inherits the QGLViewer class) will be adopted.

3.2.4 Maps and VBOs

Their is a last feature that has already been shown in a previous example (2.14) but hasn't been detailed yet: sharing between plugins through the main application. This subsection explains how to make the **CGoGN** maps created within a plugin available for other and how to attach VBOs to them. Once again, the following assumes that you are already familiar with the **CGoGN** maps and VBO implementation.

Map and VBO handling types

Maps are the main object of interest of the **CGoGN** library. Plugins are supposed to offer functionalities, and specific treatments on the geometric object provided by CGoGN. This is why it is necessary that several plugins can work on a same map.

VBOs are essential datas if you want to render maps and thus their number is limited within **OpenGL** application. This is why it is interesting to allow maps to carry their VBOs so they can also provide them to other plugins and not to let each plugin create and allocate their own VBO objects for rendering one and only map.

MapHandler: It's the type that handles CGoGN's map. It basicaly is a class that associate a reference to a generic map type (**CGoGN::GenericMap**) with a list of VBO references. Here are the constructor and the main useful methods of the **MapHandler** class:

`MapHandler(CGoGN::GenericMap* map)`

	name	type	description
argument:	map	CGoGN::GenericMap*	A reference to an allocated CGoGN generic map

`CGoGN::GenericMap* map()`

	name	type	description
return:		CGoGN::GenericMap*	A reference to the CGoGN map contained by the current MapHandler.

The VBO also possess their own handling type **VBOHandler** that will be detailed further. Here are how to access these object from a **MapHandler** instance.

`VBOHandler* findVBO(QString name)`

	name	type	description
argument:	name	QString	The name of the VBO you want to access.
return:		VBOHandler*	A reference to the VBO associate to the given name, NULL if such a VBO isn't associated with the generic map handled by this object.

`QList<VBOHandler*> findVBOsMatching(QRegExp regexp)`

	name	type	description
argument:	regexp	QRegExp	A regular expression to match the names of the VBOs you want to access.
return:		VBOHandler*	A reference to the VBO associate matching the regular expression.

`bool addVBO(VBOHandler* vboH)`

	name	type	description
argument:	vboH	VBOHandler*	A reference to an handled vbo you want want to associate with the map handled by this object.
return:		bool	A boolean whether or not the method has succeeded.

`VBOHandler* addNewVBO(QString vboName)`

	name	type	description
argument:	vboName	QString	A name for the VBO you want to create and associate with the map handled by this object. Typically the name matches this pattern: "*[function]VBO*". For example, for a position VBO the name should be "*positionVBO*", for a normal VBO "*normalVBO*".
return:		bool	A reference to the handler of the newly created VBO, NULL if the method fails, typically if a VBO with a same name as the given name already exists.

`VBOHandler* takeVBO(VBOHandler* vbo)`

	name	type	description
argument:	vbo	VBOHandler*	A pointer to the handler VBO you want to disassociate to the map handled by the current object.
return:		bool	A reference to the handler no longer associate to the map handled by the current object, NULL if failure.

`int countVBO()`

	name	type	description
return:		int	The number of VBOs associated to the map handled by the current object.

VBOHandler: Basically it's the type that handles VBO by associating them to a recognizable name. Actually this class extends **CGoGN**'s VBO class (**CGoGN::Utils::VBO**) and contains attributes such as his name and a list of reference to instances of **MapRender** class the VBO is associated with. Here are **VBOHandler**'s useful constructors and public methods you'll have to use if you want to render shared maps using VBOs:

`VBOHandler(QString name)`

	name	type	description
argument:	name	QString	The name associated to the newly created VBO this object will handle.

`VBOHandler(VBO vbo, QString name)`

	name	type	description
argument:	vbo	VBO	A CGoGN vbo that will be handled by this object
	name	QString	The name associated to the VBO this object will handle.

`bool isShared()`

	name	type	description
return:		bool	A boolean whether or not this handled VBO is shared between two or more instance of **MapHandler**.

`QString getName()`

	name	type	description
return:		QString	The name which is associated to the handled VBO.

Important note: Be sure to remember that VBO object **must** be created withing an **OpenGL** context, that is to say either within the `cb_initGL(Scene*)` method either within `cb_redraw(Scene*)`.

Sharing maps

The idea is that any plugin can create a map and then reference it into the main application that can then share it with other plugins on user's demand. The procedure within a plugin is:

1. create a **CGoGN** map
2. handle the map with a **MapHandler** class instance.
3. create and handle VBO using the **VBO** handler class.
4. associate the handled VBOs to the map
5. send the **MapHandler** object to the main application so that it will reference it and make it available for other plugins.

Here are the methods provided by **VisualPlugin** that you'll have to call in order to send a **MapHandler** object to the main application:

```
bool addReferencedMap(QString map_name, MapHandler* map)
```

	name	type	description
argument:	map_name	QString	The name under which the main application will reference the handled map.
	map	MapHandler*	The reference to handler of the map you want to made available for sharing.
return:		bool	A boolean whether or not the method succeeds. For example, failure if another map is already referenced under the same name.

`MapHandler* getReferencedMap(QString map_name)`

	name	type	description
argument:	map_name	QString	The name of the map handler referenced by the main application.
return:		MapHandler*	A pointer to the map handler referenced by the main application under the given name, NULL is failure, for example if no map handler is referenced under this name

Note: Since the map are meant to be shared manually between plugins by the user, the last method shouldn't be useful.

Plugins maps callback

In a similar way than callbacks methods are called when a user links a scene to a plugin, callbacks methods are called when maps are shared with plugins.A coder will probably want to runs specific code when the user share a map with his plugin. To do so, **VisualPlugin** provides the following callback methods that the code only have to overload:

- `void cb_recievedMap(MapHandler* map)`
- `void cb_removingMap(MapHandler* map)`

The two methods are respectively called when the user shares the map handled by the map handler given as a parameter with the current plugin.

The **VisualPlugin** class maintains a list `l_map` containing the references to all the map handlers that handle the map shared with the plugin. Be sure **not to** remove any element of this list with you own code. The only permitted operation that could be allowed are sorting or re-ordering the list.

Finally it's important that you know that the `cb_recievedMap()` is called **after** the map handler is added to `l_map` and that `cb_removingMap()` is called **before** the map handler is removed from this list.