Tulip Developer Handbook

Tulip Developer Handbook

Table of Contents

List of Examples

Chapter 1. Introduction

Tulip is a software system for the area of the Information visualization. It is becoming more and more useful for the correct analysis of existing data sets. This need results from progress in data acquisition methods, and from the huge effort made to build computer access to the human knowledge. As an example, for the human genome database, the raw data acquisition phase seems to be completed; however, to reach the ultimate goal of providing new medical treatment, it is necessary to understand these data. In such an application, the information visualization views of the data in order to explore and extend knowledge.

Here we focus on data that can be represented by a graph. In most of cases a graph structure can be extracted from existing data sets. The most well-known is the World Wide Web where links between pages can be considered as edges and pages as nodes. Another one is the human metabolism data-set where chemical reactions can be embedded in a Petri net, literature co-citations are modeled as edges between nodes of this network, and metabolic pathway are considered as clusters of the resulting graph.

Systems to visualize graphs have come to the fore during the last ten years. To our knowledge, no one provides the following capabilities simultaneously :

- Visualization and navigation in 2 or 3 dimensions
- Support of huge graphs
- Support of graph modifications
- Management of clusters
- Management of unbounded number of shared properties between graphs
- A mechanism for evaluating internal properties
- Extension and reuse without recompilation of the software
- Free of use and open source

To experiment with tools to handle graphs of the size of those induced by the human genome data set, one needs a software solution with all these capabilities. That's why we decided to build our own graph visualization software that meets these requirements. Tulip has been developed in C++, and uses two well-known libraries, OpenGL and Qt. The final program enables visualization, clustering and automatic drawing of graphs with up to 1.000.000 elements on personal computer.

This manual is an help necessary for the developers of an application using Tulip libraries and for the developers of the Tulip Team. It explains how to compile the libraries and the software, it does a presentation of the main methods available on the libraries. Chapters relate the programming and documentation guidelines.

Chapter 2. Installation

The “configure” shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a `Makefile` in each directory of the package. It may also create one or more `.h` files containing system-dependent definitions. Finally, it creates a shell script `config.status` that you can run in the future to recreate the current configuration, a file `config.cache` that saves the results of its tests to speed up reconfiguring, and a file `config.log` containing compiler output (useful mainly for debugging `configure`).

If you need to do unusual things to compile the package, please try to configure out how `configure` could check whether to do them, and mail diffs or instructions to the address given in the `README` so they can be considered for the next release. If at some point `config.cache` contains results you don’t want to keep, you may remove or edit it.

The file `configure.in` is used to create `configure` by a program called `autoconf`. You only need “configure.in” if you want to change it or regenerate `configure` using a newer version of `autoconf`.

The simplest way to compile this package is:

1. `cd` to the directory containing the package’s source code and type `./configure` to configure the package for your system. If you’re using `csh` on an old version of System V, you might need to type `sh ./configure` instead to prevent `csh` from trying to execute `configure` itself. To know all of the possible options, type `./configure --help` or read the next section about options.Running `configure` takes awhile. While running, it prints some messages telling which features it is checking for.

2. Type `make` to compile the package.

3. Optionally, type `make check` to run any self-tests that come with the package.

4. Type `make install` to install the programs, data files and documentation.

5. You can remove the program binaries and object files from the source code directory by typing `make clean`. To also remove the files that `configure` created (so you can compile the package for a different kind of computer), type `make distclean`. There is also a `make maintainer-clean` target, but that is intended mainly for the package’s developers. If you use it, you may have to get all sorts of other programs in order to regenerate files that came with the distribution.

If you don’t find the `configure` shell, you have to generate it with `gen-conf.sh`. The processing using the GNU build sytem gives a `configure` shell.

Caution

Verify `QTDIR` is properly set : `QTDIR/bin` should contain moc, qtconfig... , `QTDIR/lib` the libs and `QTDIR/include` the Qt headers ...

To compile the documentations, type `make html` and `make install` to install the documentation the `share` directory.

2.1. Options

In this section, you can find the most used options. To know all of the possible options, type `./configure --help`

`--prefix` = *value*
: The *value* is the path where you want to install Tulip. `bin, include, lib` directories was created in this location. By default, it is `/usr/local/`.

`--enable-debug`
: Add compilation flags to allow debugging.

`--enable-maintainer-mode` = value
: Enable make rules and dependencies not useful (and sometimes confusing) to the casual installer. For example, if you modify a `Makefile.am` file, it is detected and the `Makefile` is updated.

`--CXXFLAGS`
: Enable to specify special tags for the g++ compiler. It is an important part to have great performings. For example, you can specify the architecture of your PC : --CXXFLAGS="-DNDEBUG -O3 --funroll-loops `-mtune=pentium4 -narch=pentium4` -pipe". For more informations about the options of compilations for g++, see the web site of gcc, here[1].

[1] http://gcc.gnu.org/onlinedocs/gcc-4.0.1/gcc/Submodel-Options.html#Submodel-Options

Chapter 3. Tulip Library

3.1. Introduction

Efficient visualization of graphs and their usage for data analysis implies

- the manipulation of the structure of the graph
- the extraction of parts of it
- association of values to a graph's elements (nodes and edges)
- the computation of intrinsic parameters (parameters derived from a graph's structure, e.g. in a filesystem graph (directory tree): number of files in a directory, can be computed by counting outgoing edges)
- the computation of extrinsic parameters (parameters derived from external information, e.g. in a filesystem graph: file size)

This chapter describes the Tulip data structure that takes into account all the requirement of a graph visualization system. For each part we describe the general principle and then we give examples explaining how to do it with the Tulip library.

3.2. Graphs

The core of the Tulip library provides an interface for the manipulation of graphs. It enables one to access and modify the structure of a graph. The aim of this library is to be as general as possible and thus it manipulates a general class of graphs called directed pseudo-graphs. In a pseudo graph, there can be more than one edge between two nodes, and loops are permitted. A loop is an edge that links a node to itself. Furthermore, edges are directed, thus an edge u->v is distinct from an edge v->u.

Because we use pseudo-graphs, there can be more than one edge u->v, so it is not possible to distinguish two edges using only source and target (u,v). To make this possible, all the elements in Tulip are entities (C++ objects). Thus, even if two edges have the same source and the same target, they are distinct.

The elements of a graph are encapsulated in the graph. It is therefore not possible to access the graph's structure through elements, all operations must be done by querying the graph. For example, to know the source of an edge e of graph G, one must ask G, not e, what e's source is. This makes the use of the library less intuitive, but it minimizes memory usage for entities and allows to share them between subgraphs. Building a container of elements is cheap, because to handle elements, Tulip uses objects which use the same amount of storage as integers.

The library supports access and modification of the graph structure. The access to the structure are made by using iterators, one very important point is that the iterator are not persistent. Thus, if one modify the graph structure all the iterators on the graph structure can be invalid. This property enables to prevent from cloning the data structure and thus enables better access to it. For ease of use, Tulip includes mechanism that enables to transform an iterator into stable iterator, one must keep in mind that it corresponds to clone the data structure and thus, it should be use only if it is necessary.

If one uses Tulip only for the manipulation of one graph (no graph hierarchy), the list of available operations on the graph is given afterward. In the next section we will enhance the set of operations and the actions that they perform in order to manage a hierarchy of subgraphs

List of available modification operations

- `node addNode()` : creates a new node in the graph and returns its identifier.
- `edge addEdge(node,node)` : create a new edge in the graph, given the source and target.
- `void delNode(node)` : deletes the given node.
- `void delEdge(edge)` : deletes the given edge.
- `void reverse(edge)` : reverses an edge (swaps source and target).

List of available access operations

- `unsigned int deg(node)` : returns the degree of a node (number of edges).
- `unsigned int indeg(node)` : returns the in degree of a node (number of times it is a target).
- `unsigned int outdeg(node)` : returns the out degree of a node (number of times it is a source).
- `node source(edge)` : returns the source of an edge.
- `node target(edge)` : returns the target of an edge.
- `node opposite(edge,node)` : it enables to obtain the opposite of a node of an edge.
- `Iterator * getInNodes(node)` : returns an iterator on the predecessor nodes of a node.
- `Iterator * getOutNodes(node)` : returns an iterator on the successor nodes of a node.
- `Iterator * getInOutNodes(node)` : returns an iterator on the neighbor nodes of a node.
- `Iterator * getInEdges(node)` : returns an iterator on the predecessor edges of a node.
- `Iterator * getOutEdges(node)` : returns an iterator on the successor edges of a node.
- `Iterator * getInOutEdges(node)` : returns an iterator on the neighbor edges of a node.

3.3. Hierarchy of graphs

The Tulip library can also manage subgraphs. By definition a subgraph G’ of a graph G is part (subset) of the elements of G such that G’ is also a graph (all sources and targets of the edges of G’ must be in G’). As a subgraph is also a graph it can itself contain subgraphs. In such a hierarchy, if a graph G" is a descendant of a graph G, G" is also a subgraph of G.

One of the strong point of Tulip is to ensure efficiently that all elements are shared between graphs in a hierarchy of graphs. Thus, if a node n is an element of a graph G and of a graph G’, the entity n is the same in both graphs. Of course, the parameters of the entity can change between graphs. For instance, the degree of n can be smaller for a subgraph, as it can have less edges.

The subgraph relation in the hierarchy is preserved when one modifies a graph. This means that if one adds a node to a graph, this node is automatically added to all its ancestors as well. If one deletes a node, this node is automatically deleted from all the descendants of the graph. If one reverses an edge, this edge is reversed in all the graphs of the hierarchy.

In order to manipulate a hierarchy of graphs, more functions have been added to those introduced above. They provide navigation and modification for the hierarchy. The access to the hierarchy is provided by iterators, which are not persistent and thus, if the hierarchy is modified, the iterators are invalid.

List of available modification operations

- `Graph *addSubGraph()` : returns an empty subgraph of this graph.
- `Graph *delSubGraph(Graph *)` : deletes a subgraph. Its descendants continue to be descendants of this graph.
- `Graph *delAllSubGraph(Graph *)` : deletes a subgraph and all its descendants.
- `edge addEdge(edge)` : adds an edge element from another graph in the hierarchy.
- `void addNode(node)` : adds a node element from another graph in the hierarchy.

List of available access operations

- `Iterator * getSubGraphs()` : returns an iterator on the subgraphs.
- `Graph * getSuperGraph()` : returns the parent of the graph. If the graph has no parent, it returns the graph itself.

3.4. Attributes

An attributes is a kind of property that can be associated to a graph. An attributes has a name (a string) and a value of any type. It can be, for example ,the name of a graph, or a date of creation of the graph.

Attributes can be added and accessed with those three following member functions :

- `const DataSet getAttributes()` : returns the attributes of a graph.
- `template<typename ATTRIBUTETYPE>bool getAttribute(const std::string &name, ATTRIBUTETYPE &value)` : get an attribute.
- `template<typename ATTRIBUTETYPE>void setAttribute (const std::string &name, const ATTRIBUTETYPE &value)` : set a new attribute value.

3.5. Properties

In Tulip, a property is an attribute of an element of a graph. It is called a property in order to prevent confusion with attributes of a graph: properties are for elements and attributes are for graphs. In Tulip, a property is always defined for both kinds of elements (nodes and edges), so one can always query for the value of the property associated with any edge or node.

To access the value of an elements one must query the graph for a property. This makes the use of the library less intuitive, but it minimizes memory usage for properties.

A property can be seen as an associative table where you can set and get the value for every element. All property operations have a TYPE argument, so there is no need to cast the result of a property query. The standard operations of a property are:

List of available modification operations

- `void setNodeValue(node,TYPE)` : sets the value of a node.
- `void setAllNodeValue(TYPE)` : sets the value of all nodes.
- `void setEdgeValue(edge,TYPE)` : sets the value of an edge.
- `void setAllEdgeValue(TYPE)` : sets the value of all edges.

List of available access operations

- `TYPE getNodeValue(node)` : returns the value of a node.
- `TYPE getEdgeValue(edge)` : returns the value of an edge.

For each property type there is a specific implementation (subclass) that allows operations which are specific to the property type (see Tulip libraries documentation). For instance, it is possible to obtain the maximum value of a property if the property type is `double`.

A graph includes a set of functions that enables to obtain/create/delete a property. Because the C++ signature of functions does not include the return type, the syntax for this call is not very simple. For instance, if one wants to obtain a property containing double (called DoubleProperty in Tulip) one must use the following syntax : `DoubleProperty *metric=graph->getProperty<DoubleProperty>("name of the property");` In the graph each property is identified by its name which is a std::string, when one asks for a property the type of this property is checked using the run time type interrogation mechanism of C++. Warning: This test only happens when one compiles its sources in DEBUG mode (default mode). In order to facilitate the navigation/edition of the set of properties, a set of functions is accessible through the graph interface.

List of available operations

- `Iterator * getLocalProperties()` : returns an iterator on all properties of this graph.
- `void delLocalProperty(const std::string&)` : deletes a property.
- `bool existLocalProperty(const std::string&)` : returns true if the property exists.
- `PropertyType * getLocalProperty (const std::string&)` : returns the property. If it did not exist, creates it first

For the property mechanism described above to work with a hierarchy of graphs, a mechanism have been added to share properties between graphs, which works like this: if a property exists in an ancestor of a graph G, it also exists in the graph G. Thus, properties of graphs are inherited like members of objects in object-oriented languages. In order to facilitate the navigation/edition of properties, a set of function is accessible through the graph interface.

List of available operations

- `Iterators * getInheritedProperties()` : returns an iterator on all properties (both inherited and local).
- `bool existProperty(const std::string&)` : returns true if the property exists (inherited or local).
- `PropertyType * getProperty(const std::string&)` : returns the property (inherited or local). If it did not exist, creates it first (locally)

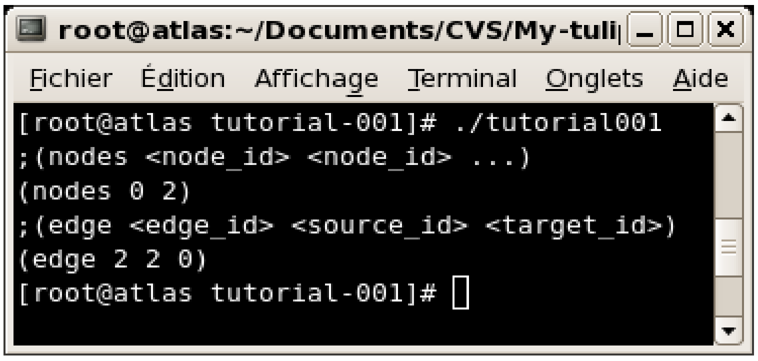
3.6. TUTORIAL Intro.

Before doing any of the tutorials please read the following warnings :

- Follow the tutorials, one by one, from the first one to the last one.
- You can find at the end of each tutorial, the integral source code that we used.
- If you want more details on a specific function or class please read the Tulip Graph library documentation[1].

3.7. TUTORIAL 001 : Graphs creation, adding and deleting nodes or edges.

In this first tutorial, we will show you how to create a graph, add three nodes and three edges, remove an edge and a node, and finally, print the result on the standard output.


root@atlas:~/Documents/CVS/My-tuli
Fichier Édition Affichage Terminal Onglets Aide

```
[root@atlas tutorial-001]# ./tutorial001
;(nodes <node_id> <node_id> ...)
(nodes 0 2)
;(edge <edge_id> <source_id> <target_id>)
(edge 2 2 0)
[root@atlas tutorial-001]#
```

3.7.1. 1. Header files

Let's start with the files we need to include :

```
#include <iostream>
#include <tulip/Graph.h>
```

1 ../../doxygen/tulip.html

• `iostream` : This "file" contains the C++ standard declarations for in and out streams. We need it in this tutorial to print the final graph on the standard output.

• `tulip/Graph.h` : This file is the core of the tulip graph API. It provides declarations for graphs (edges , nodes) and functions to load one from a file, to save one, and a lot more. You can find a list in the Tulip Graph library documentation[2].

The namespaces usage declarations, so that we can omit namespace prefix.

```
using namespace std;
using namespace tlp;
```

3.7.2. 2. Creation of a Graph

Create an empty graph with the function `Graph* tlp::newGraph( )`. This function returns a pointer on a empty Graph.

```
int main() {
  //create an empty graph
  Graph *graph = tlp::newGraph();
```

3.7.3. 3. Add nodes

In the following, we are adding three nodes with the member function `node Graph::addNode ()` that will, create an instance of a 'node', add it to the graph, and return it.

Note

Using this function, the node is also added in all the graph ancestors to maintain the sub-graph relation between graphs.

```
  //add three nodes
  node n1 = graph->addNode();
  node n2 = graph->addNode();
  node n3 = graph->addNode();
```

3.7.4. 4. Add edges

Now that nodes are created, we can create the edges. To do so, we can use the function `edge Graph::addEdge ( const node, const node )` that will, add a new edge in the graph and return it.

2 ../../doxygen/tulip-lib/Graph_8h.html

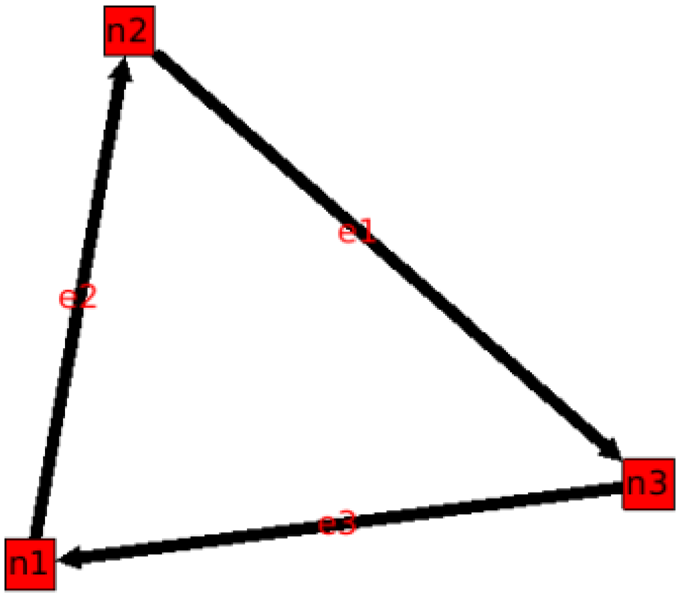
Note

The edge is also added in all the super-graph of the graph to maintain the sub-graph relation between graphs.

The first parameter is the "source node", and, of course, the second is the "target node" (in tulip, every edge are oriented but you can choose not to consider the orientation). We will see later (TUTORIAL 005) that the edges enumeration order is the one in which they are added.

```
//add three edges
edge e1 = graph->addEdge(n2,n3);
edge e2 = graph->addEdge(n1,n2);
edge e3 = graph->addEdge(n3,n1);
```

Following is a picture of the graph that we just have created. It is being displayed with tulip.



3.7.5. 5. Delete an edge and a node

The Graph class provides member functions to delete edges and nodes.

- `void tlp::Graph::delEdge (const edge)` : delete an edge of the graph. This edge is also removed in all the sub-graphs hierarchy to maintain the sub-graph relation between graphs. The ordering of edges is preserved.

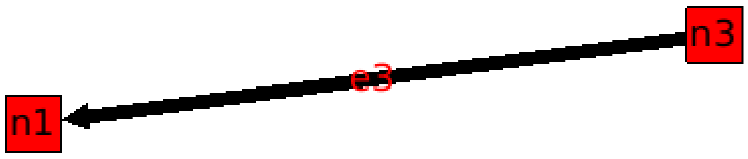
- `void tlp::Graph::delNode ( const node )` : delete a node of the graph. This node is also removed in all the sub-graph of the graph to maintain the sub-graph relation between graphs. When the node is deleted, all its edges are deleted (in and out edges).

Note

The class Graph implements member functions like `void delAllNode (const node)`, and, `void delAllEdge (const edge).`

```
//delete an edge
graph->delEdge(e1);

//delete a node
graph->delNode(n2);
```

Following is our graph with node n2 deleted.



3.7.6. 6. Printing the graph

The class graph has a friend function which is an overload of the stream operator <<. This function will print the graph (only nodes and edges) in an output stream (here, the standard output, "cout"), in the tulip format.

```
//print the result on the standard output
cout << graph << flush;
```

3.7.7. 7. Saving a graph

Instead of having our graph printed on the standard output, we can save it in a .tlp (tulip format) suffixed file that can be read by tulip :

```
//Save  the graph :
tlp::saveGraph(graph,"tuto1.tlp");
```

3.7.8. 8. Graph deletion

Before exiting the main function, do not forget memory leaks, and delete the graph to free memory usages.

```
  //delete the graph
  delete graph;
  return EXIT_SUCCESS;
}
```

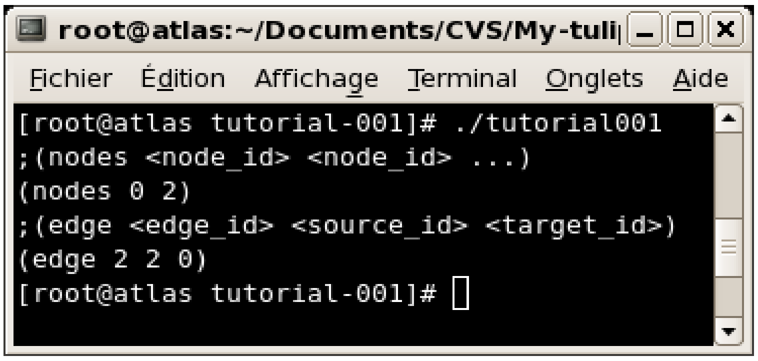
3.7.9. 9. Compiling and running the program.

Compile this program with this command :

```
g++ `tulip-config --libs --cxxflags` tutorial.cpp -o tutorial001
```

Run it to have a look :

```
./tutorial001
```


root@atlas:~/Documents/CVS/My-tuli
Fichier Édition Affichage Terminal Onglets Aide

```
[root@atlas tutorial-001]# ./tutorial001
;(nodes <node_id> <node_id> ...)
(nodes 0 2)
;(edge <edge_id> <source_id> <target_id>)
(edge 2 2 0)
[root@atlas tutorial-001]#
```

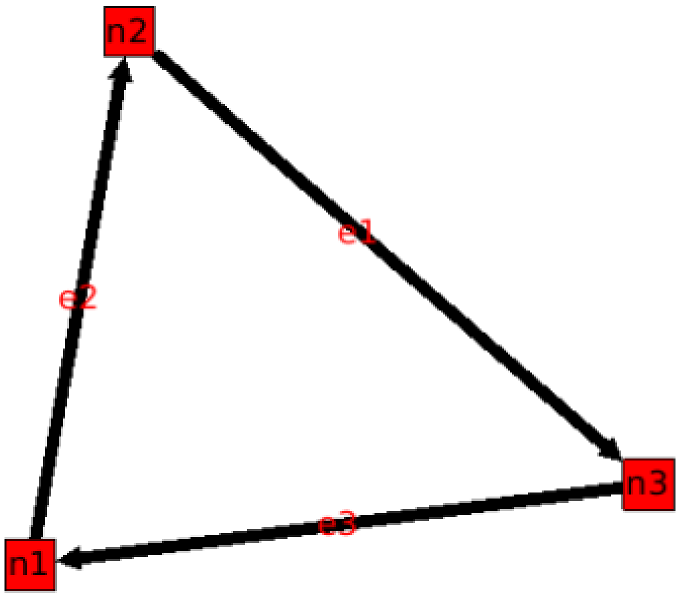
3.7.10. 10. Source Code

```
#include <iostream>
#include <tulip/Graph.h>

/**
 *
 * Tutorial 001
 *
 * Create a graph
 * add three nodes and three edges
 * remove an edge and a node
 * print the result on the standard output
 *
 */

using namespace std;
using namespace tlp;

int main() {
  //create an empty graph
  Graph *graph = tlp::newGraph();

  //add three nodes
  node n1 = graph->addNode();
  node n2 = graph->addNode();
  node n3 = graph->addNode();

  //add three edges
  edge e1 = graph->addEdge(n2,n3);
  edge e2 = graph->addEdge(n1,n2);
  edge e3 = graph->addEdge(n3,n1);

  //delete an edge
  graph->delEdge(e1);
  //delete a node
  graph->delNode(n2);

  //print the result on the standard output
  cout << graph << flush ;

  tlp::saveGraph(graph,"tuto1.tlp");

  //delete the graph
  delete graph;
  return EXIT_SUCCESS;
}
```

3.8. TUTORIAL 002 : Iterating over a graph (class Iterator and the macro forEach)

In this tutorial, we will, display on the standard output, all the structure using iterators. For each node, we will display its ancestors, successors, neighbors, and, its incoming and outgoing edges.

In this tutorial, the graph created is the same that in Tutorial 1 (after the 3 edges were added) see the following picture :



3.8.1. 1. Header files (Same as Tutorial 1)

Let's start with the files we need to include :

```
#include <iostream>
#include <tulip/Graph.h>
```

- `iostream` : This "file" contains the C++ standard declarations for in and out streams. We need it in this tutorial to print the final graph on the standard output.

- `tulip/Graph.h` : This file is the core of the tulip graph API. It provides declarations for graphs (edges , nodes) and functions to load one from a file, to save one, and a lot more. You can find a list in the doxygen documentation[3].

As you can see, we just need the "Graph.h" header file to create a graph and iterate over its nodes, even though the declaration of the abstract class "Iterator" is in Iterator.h

3.8.2. 2. Iterating over all nodes

To iterate over all nodes, we need to create an Iterator on the graph nodes with the member function `Iterator* Graph::getNodes () const`, we will make it point on the graphs nodes.

```
  Iterator<node> *itNodes = graph->getNodes();
```

The documentation of the interface Iterator can be found here.[4]

With the functions `template <class itType > bool tlp::Iterator< itType >::hasNext ( )` and `node next ( )`, we can iterate through our graph nodes with a simple while :

```
   while(itNodes->hasNext()) {
    node n = itNodes->next();
```

In this `while` loop, we display some node topological properties :

```
    cout << "node: " <<  n.id << endl;
    cout << " degree: " << graph->deg(n) << endl;
    cout << " in-degree: " << graph->indeg(n) << endl;
    cout << " out-degree: " << graph->outdeg(n) << endl;
```

At the end of the loop, we will need to delete the iterator: `delete itNodes;`

Following is the output of this simple while loop :

```
[root@atlas tutorial-002]# ./tutorial002
```

[3] ../../doxygen/tulip-lib/Graph_8h.html
[4] http://tulip.labri.fr/doxygen/tulip-lib/structtlp_1_1Iterator.html

```
node: 0
 degree: 2
 in-degree: 1
 out-degree: 1
node: 1
 degree: 2
 in-degree: 1
 out-degree: 1
node: 2
 degree: 2
 in-degree: 1
 out-degree: 1
```

3.8.3. 3. Iterating through a node predecessors

To iterate through predecessors of node, we use the same type of Iterator, but, instead of using the function getNodes() of the class Graph, we will use the function `Iterator<node>* getInNodes (const node) const` that will return an iterator on the predecessors of a node.

```
//===========================
//iterate all predecessors of a node
cout << " predecessors: {";
Iterator<node> *itN=graph->getInNodes(n);
while(itN->hasNext()) {
  cout << itN->next().id;
  if (itN->hasNext()) cout << ",";
} delete itN; //!!!Warning : do not forget to delete iterators (memory leak)

cout << "}" << endl;
```

3.8.4. 4. Iterating through a node successors

To iterate through successors of a node, we just need to use the function `Iterator<node>* Graph::getOutNodes (const node) const` to have an Iterator on its successors.

```
//===========================
//iterate all successors of a node
cout << " successors: {";
itN = graph->getOutNodes(n);
while (itN->hasNext()) {
  cout << itN->next().id;
  if (itN->hasNext()) cout << ",";
} delete itN; //!!!Warning : do not forget to delete iterators (memory leak)

cout << "}" << endl;
```

3.8.5. 5. Iterating through a node neighbors (predecessors and successors)

For neighbors, we will use the function `Iterator<node>* Graph::getInOutNodes (const node) const` to have an Iterator on its neighbors.

```
//===========================
//iterate the neighborhood of a node
cout << " neighborhood: {";
itN = graph->getInOutNodes(n);
while(itN->hasNext()) {
  cout << itN->next().id;
  if (itN->hasNext()) cout << ",";
} delete itN; //!!!Warning : do not forget to delete iterators (memory leak)

cout << "}" << endl;
```

3.8.6. 6. Iterating through a node incoming edges

For incoming edges, we will use an Iterator on edges with the member function `Iterator<edge>* Graph::getInEdges (const node) const`.

```
//===========================
//iterate the incoming edges
cout << " incoming edges: {";
Iterator<edge> *itE=graph->getInEdges(n);
while(itE->hasNext()) {
  cout << itE->next().id;
  if (itE->hasNext()) cout << ",";
} delete itE; //!!!Warning : do not forget to delete iterators (memory leak)

cout << "}" << endl;
cout << " outcoming edges: {";
```

3.8.7. 7. Iterating through a node outgoing edges

For outgoing edges, we will use the function `Iterator<edge>* Graph::getOutEdges (const node) const`.

```
//===========================
//iterate the outcomming edges
itE = graph->getOutEdges(n);
while(itE->hasNext()) {
  cout << itE->next().id;
  if (itE->hasNext()) cout << ",";
} delete itE; //!!!Warning : do not forget to delete iterators (memory leak)

cout << "}" << endl;
```

3.8.8. 8. Iterating through a node adjacent edges

For adjacent edges, we will use the function `Iterator<edge>* Graph::getInOutEdges (const node) const`.

```
//===========================
//iterate the adjacent edges
cout << " adjacent edges: {";
itE = graph->getInOutEdges(n);
while(itE->hasNext()) {
  cout << itE->next().id;
  if (itE->hasNext()) cout << ",";
} delete itE; //!!!Warning : do not forget to delete iterators (memory leak)

cout << "}" << endl;
```

3.8.9. Don't forget memory leaks

As we are still in the first while (iterating through all nodes) we need to delete the Iterator on Nodes :

```
}// end while
delete itNodes; //!!!Warning : do not forget to delete iterators (memory leak)
```

3.8.10. 9. Iterating on edges (all edges).

Some times it can be useful to iterate on edges, for example in the algorithm of Kruskal. That is why the graph class owns the function `Iterator<edge>* Graph::getEdges (const node) const`, that return a pointer on an Iterator of type edge. Following is an exemple of its use.

```
//===========================
//Iterate all edges
Iterator<edge> *itEdges=graph->getEdges();
while(itEdges->hasNext()) {
  edge e = itEdges->next();
  cout << "edge: " << e.id;
  cout << " source: " << graph->source(e).id;
  cout << " target: " << graph->target(e).id;
  cout << endl;
} delete itEdges; //!!!Warning : do not forget to delete iterators (memory leak)
```

3.8.11. 10. The forEach Macro

To simplify the use of Iterators, the API of tulip provides a macro forEach which is quite similar to the foreach of C# or Java. It takes two parameters :

- A variable :
- An Iterator for the same type as the variable, for example : Variable of type node, Graph::getNodes().

Warning

- Must use breakForEach to break iteration
- Must use returnForEach to return during iteration

This macro function is defined in the header file : tulip/ForEach.h

Following is a small example of its use.

```
#include <tulip/ForEach.h>

  //...
  //main
  //load Graph
  //...

  node n = graph->getOneNode();
  cout << "In Edges :" << endl;
  edge e;
  forEach(e, graph->getInEdges(n))
  {
    cout << e.id << ",";
  }

  //...
```

3.8.12. Source Code

```
#include <iostream>
#include <tulip/Graph.h>

/**
 * Tutorial 002
 *
 * Create a graph
 * display all the structure using iterators
 *
 */

using namespace std;
using namespace tlp;

void buildGraph(Graph *graph) {
  //add three nodes
  node n0=graph->addNode();
  node n1=graph->addNode();
  node n2=graph->addNode();
  //add three edges
  graph->addEdge(n1,n2);
  graph->addEdge(n0,n1);
  graph->addEdge(n2,n0);
}
```

```
int main() {
  //create an empty graph
  Graph *graph=tlp::newGraph();

  //build the graph
  buildGraph(graph);

  //===========================
  //Iterate all nodes and display the structure
  Iterator<node> *itNodes = graph->getNodes();
  while(itNodes->hasNext()) {
    node n = itNodes->next();
    cout << "node: " <<  n.id << endl;
    cout << " degree: " << graph->deg(n) << endl;
    cout << " in-degree: " << graph->indeg(n) << endl;
    cout << " out-degree: " << graph->outdeg(n) << endl;

    //===========================
    //iterate all ancestors of a node
    cout << " ancestors: {";
    Iterator<node> *itN=graph->getInNodes(n);
    while(itN->hasNext()) {
      cout << itN->next().id;
      if (itN->hasNext()) cout << ",";
    } delete itN; //!!!Warning : do not forget to delete iterators (memory leak)

    cout << "}" << endl;

    //===========================
    //iterate all successors of a node
    cout << " successors: {";
    itN = graph->getOutNodes(n);
    while (itN->hasNext()) {
      cout << itN->next().id;
      if (itN->hasNext()) cout << ",";
    } delete itN; //!!!Warning : do not forget to delete iterators (memory leak)

    cout << "}" << endl;

    //===========================
    //iterate the neighborhood of a node
    cout << " neighborhood: {";
    itN = graph->getInOutNodes(n);
    while(itN->hasNext()) {
      cout << itN->next().id;
      if (itN->hasNext()) cout << ",";
    } delete itN; //!!!Warning : do not forget to delete iterators (memory leak)

    cout << "}" << endl;

    //===========================
    //iterate the incoming edges
    cout << " incoming edges: {";
    Iterator<edge> *itE=graph->getInEdges(n);
    while(itE->hasNext()) {
      cout << itE->next().id;
      if (itE->hasNext()) cout << ",";
    } delete itE; //!!!Warning : do not forget to delete iterators (memory leak)

    cout << "}" << endl;
    cout << " outcoming edges: {";

    //===========================
    //iterate the outcomming edges
    itE = graph->getOutEdges(n);
    while(itE->hasNext()) {
      cout << itE->next().id;
      if (itE->hasNext()) cout << ",";
    } delete itE; //!!!Warning : do not forget to delete iterators (memory leak)

    cout << "}" << endl;

    //===========================
    //iterate the adjacent edges
    cout << " adjacent edges: {";
    itE = graph->getInOutEdges(n);
    while(itE->hasNext()) {
      cout << itE->next().id;
      if (itE->hasNext()) cout << ",";
    } delete itE; //!!!Warning : do not forget to delete iterators (memory leak)

    cout << "}" << endl;

  } delete itNodes; //!!!Warning : do not forget to delete iterators (memory leak)


  //===========================
  //Iterate all edges
  Iterator<edge> *itEdges=graph->getEdges();
  while(itEdges->hasNext()) {
    edge e = itEdges->next();
    cout << "edge: " << e.id;
    cout << " source: " << graph->source(e).id;
    cout << " target: " << graph->target(e).id;
    cout << endl;
  } delete itEdges; //!!!Warning : do not forget to delete iterators (memory leak)


  delete graph; //delete the entire graph
  return EXIT_SUCCESS;
}
```

3.9. TUTORIAL 003 : Properties

This tutorial will show you how to add / create properties to a Graph. For local or inherited properties, see tututorial 005. An instance of a property is owned by a graph and is an association table between the elements of graph (nodes and edges) and values of a predefined type.

3.9.1. 1. Header files and predefined properties

In tulip API, every type of property is declared in its own header file. Following is a list of those header files and the type of value which can be associated to an element of the graph:

- DoubleProperty : tulip/DoubleProperty.h / value type for edge = double, node = double
- BooleanProperty : tulip/BooleanProperty.h / value type for edge = bool, node = bool
- IntegerProperty: tulip/IntegerProperty.h / value type for edge = int, node = int
- LayoutProperty : tulip/LayoutProperty.h / value type for edge = Coord(), node = vector<Coord>()
- ColorProperty : tulip/ColorProperty.h / value type for edge = Color(), node = Color()
- SizeProperty : tulip/SizeProperty.h / value type for edge = Size(), node = Size()
- StringProperty : tulip/StringProperty.h / value type for edge = string, node = string
- GraphProperty : tulip/GraphProperty.h / value type for edge = graph, node = graph

3.9.2. 2. Creation of a property.

The creation of a property is accomplished by the function `Graph::getLocalProperty<TypeProperty>("n of the property")`. This function returns a pointer to a property. The real type of the property is given with the template parameter. If the property of the given name does not yet exists, a new one is created and returned.

Warning

Using of delete on that property will cause a segmentation violation (use delLocalProperty instead).

Following is a sample of code that creates 8 properties :

```
//Get and create several properties
DoubleProperty *metric = graph->getLocalProperty<DoubleProperty>("firstMetric")

BooleanProperty *select = graph->getLocalProperty<BooleanProperty>("firstSelect

LayoutProperty *layout = graph->getLocalProperty<LayoutProperty>("firstLayout")

IntegerProperty *integer = graph->getLocalProperty<IntegerProperty>("firstInteg

ColorProperty *colors = graph->getLocalProperty<ColorProperty>("firstColors");

SizeProperty *sizes = graph->getLocalProperty<SizeProperty>("firstSizes");

GraphProperty *meta = graph->getLocalProperty<GraphProperty>("firstMeta");

StringProperty *strings = graph->getLocalProperty<StringProperty>("firstString"
```

3.9.3. 3. Initialize all properties.

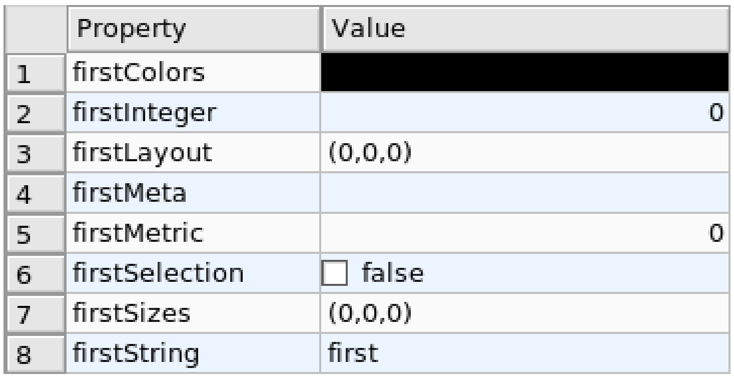
One property has to be initialized for both edges and nodes. It is done with the functions `setAllNodeValue(value)` and `setAllEdgeValue(value)` which are both member functions of the property.

Following is an example :

```
//initialize all the properties
```

```
metric->setAllNodeValue(0.0);
metric->setAllEdgeValue(0.0);
select->setAllNodeValue(false);
select->setAllEdgeValue(false);
layout->setAllNodeValue(Coord(0,0,0)); //coordinates
layout->setAllEdgeValue(vector<Coord>());//Vector of bends
integer->setAllNodeValue(0);
integer->setAllEdgeValue(0);
sizes->setAllNodeValue(Size(0,0,0)); //width, height, depth
sizes->setAllEdgeValue(Size(0,0,0)); //start_size, end_size, arrow_size

colors->setAllNodeValue(Color(0,0,0,0));//Red, green, blue, alpha
colors->setAllEdgeValue(Color(0,0,0,0));//Red, green, blue, alpha
strings->setAllNodeValue("first");
strings->setAllEdgeValue("first");
meta->setAllNodeValue(graph); //an existing graph
```

Following is the display (in the tulip GUI) of the list of a node associated values for the properties previously created :



	Property	Value
1	firstColors	■
2	firstInteger	0
3	firstLayout	(0,0,0)
4	firstMeta	
5	firstMetric	0
6	firstSelection	☐ false
7	firstSizes	(0,0,0)
8	firstString	first

3.9.4. 4. Iterating over properties.

Once again, iteration is made with Iterators. The class graph has a member function `Iterator< std::string >* getLocalProperties ()` that returns an iterator on the local properties.

Following is an example :

```
cout << "List of the properties present in the graph:" << endl;
Iterator<string> *it=graph->getLocalProperties();
while (it->hasNext()) {
  cout << it->next() << endl;
} delete it;
```

You can also use the macro forEach.

3.9.5. Source Code

```
#include <iostream>
#include <tulip/BooleanProperty.h>
```

```
#include <tulip/ColorProperty.h>
#include <tulip/DoubleProperty.h>
#include <tulip/GraphProperty.h>
#include <tulip/IntegerProperty.h>
#include <tulip/LayoutProperty.h>
#include <tulip/SizeProperty.h>
#include <tulip/StringProperty.h>

/**
 * Tutorial 003
 *
 * Create a graph and a properties of each type
 * And display properties present in the graph
 */

using namespace std;
using namespace tlp;

void buildGraph(Graph *graph) {
  //add three nodes
  node n1=graph->addNode();
  node n2=graph->addNode();
  node n3=graph->addNode();
  //add three edges
  graph->addEdge(n2,n3);
  graph->addEdge(n1,n2);
  graph->addEdge(n3,n1);
}

int main() {
  //create an empty graph
  Graph *graph=tlp::newGraph();
  //build the graph
  buildGraph(graph);

  //Get and create several properties
  DoubleProperty *metric=graph->getLocalProperty<DoubleProperty>("firstMetric");

  BooleanProperty *select=graph->getLocalProperty<BooleanProperty>("firstSelecti

  LayoutProperty *layout=graph->getLocalProperty<LayoutProperty>("firstLayout");

  IntegerProperty *integer=graph->getLocalProperty<IntegerProperty>("firstInteger

  ColorProperty *colors=graph->getLocalProperty<ColorProperty>("firstColors");

  SizeProperty *sizes=graph->getLocalProperty<SizeProperty>("firstSizes");

  GraphProperty *meta=graph->getLocalProperty<GraphProperty>("firstMeta");

  StringProperty *strings=graph->getLocalProperty<StringProperty>("firstString");


  //initialize all the properties
  metric->setAllNodeValue(0.0);
  metric->setAllEdgeValue(0.0);
  select->setAllNodeValue(false);
  select->setAllEdgeValue(false);
  layout->setAllNodeValue(Coord(0,0,0)); //coordinates
  layout->setAllEdgeValue(vector<Coord>());//Vector of bends
  integer->setAllNodeValue(0);
  integer->setAllEdgeValue(0);
  sizes->setAllNodeValue(Size(0,0,0)); //width, height, depth
  sizes->setAllEdgeValue(Size(0,0,0)); //start_size, end_size, arrow_size

  colors->setAllNodeValue(Color(0,0,0,0));//Red, green, blue
  colors->setAllEdgeValue(Color(0,0,0,0));//Red, green, blue
  strings->setAllNodeValue("first");
  strings->setAllEdgeValue("first");
  meta->setAllNodeValue(graph); //an existing graph
  cout << "List of the properties present in the graph:" << endl;
  Iterator<string> *it=graph->getLocalProperties();
  while (it->hasNext()) {
    cout << it->next() << endl;
  } delete it;

  tlp::saveGraph (graph, "tutoproper.tlp");
  delete graph;
  return EXIT_SUCCESS;
}
```

3.10. TUTORIAL 004 : Create your first subgraph.

This tutorial will teach you how to create subgraphs. At the end of it, we will have a hierarchy of 3 graphs.

Before anything consider the following function that creates 3 nodes and 3 edges (same as in tutorial 001):

```
void buildGraph(Graph *graph)
{
  //add three nodes
  node n1=graph->addNode();
  node n2=graph->addNode();
  node n3=graph->addNode();
  //add three edges
  graph->addEdge(n2,n3);
  graph->addEdge(n1,n2);
  graph->addEdge(n3,n1);
}
```

The creation of a subgraph is quite simple. You just have to use the function `Graph* addSubGraph (BooleanProperty* selection = 0)` . It will create and return a new SubGraph of the graph. The elements of the new subgraph are those selected in the selection(selection associated value equals true); if there is no selection an empty subgraph is returned.

In the following sample we create 3 empty subgraphs :

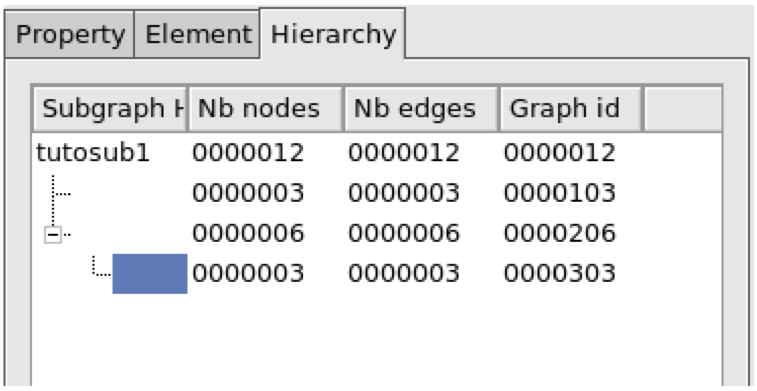
```
  //build three empty subgraphs
```

```
Graph *subgraph0=graph->addSubGraph();
Graph *subgraph1=graph->addSubGraph();
Graph *subgraph2=subgraph1->addSubGraph();
```

We now need to create some nodes and edges :

```
 //add node inside subgraphs
buildGraph(subgraph0);
buildGraph(subgraph1);
buildGraph(subgraph2);
```

Following is the hierarchy we have just created, displayed with tulip :



Subgraph H	Nb nodes	Nb edges	Graph id
tutosub1	0000012	0000012	0000012
	0000003	0000003	0000103
	0000006	0000006	0000206
	0000003	0000003	0000303

We can check that by iterating on our graph's subgraphs using the function `Iterator< Graph * >* Graph::getSubGraphs()` :

```
//iterate subgraph (0 and 1 normally ) and output them
Iterator<Graph *> *itS=graph->getSubGraphs();
while (itS->hasNext())
  cout << itS->next() << endl;
delete itS;
```

3.10.1. Source Code

```
#include <iostream>
#include <tulip/Graph.h>

/**
 * Tutorial 004
 *
 * Create a graph and three subgraphs,
 * display all the structure using iterators
 */

using namespace std;
using namespace tlp;
```

```
void buildGraph(Graph *graph) {
  //add three nodes
  node n1=graph->addNode();
  node n2=graph->addNode();
  node n3=graph->addNode();
  //add three edges
  graph->addEdge(n2,n3);
  graph->addEdge(n1,n2);
  graph->addEdge(n3,n1);
}

int main() {
  //create an empty graph
  Graph *graph=tlp::newGraph();

  //build the graph
  buildGraph(graph);

  //build two empty subgraph
  Graph *subgraph0=graph->addSubGraph();
  Graph *subgraph1=graph->addSubGraph();
  Graph *subgraph2=subgraph1->addSubGraph();

  //add node inside subgraphs
  buildGraph(subgraph0);
  buildGraph(subgraph1);
  buildGraph(subgraph2);

  //iterate subgraph (0 and 1 normally ) and output them
  Iterator<Graph *> *itS=graph->getSubGraphs();
  while (itS->hasNext())
    cout << itS->next() << endl;
  delete itS;

  delete graph;
  return EXIT_SUCCESS;
}
```

3.11. TUTORIAL 005 : Properties and subgraphs

In this tutorial, we will show you how to use properties with subgraphs, how to deal with properties in a big hierarchy. To do so, we will create a graph with some properties, several subgraphs with other properties and iterate over local and inherited properties.

3.11.1. 1. Introduction

We will first begin with the creation of the graph and its properties :

```
int main() {
  //create an empty graph
  Graph *graph=tlp::newGraph();
```

```
//build the graph
buildGraph(graph);

//Get and create several properties
BooleanProperty *select=graph->getLocalProperty<BooleanProperty>("firstSelectio

graph->getLocalProperty<ColorProperty>("firstColors");
graph->getLocalProperty<DoubleProperty>("firstMetric");

//init the selection in order to use it for building clone subgraph
select->setAllNodeValue(true);
select->setAllEdgeValue(true);
```

Note

The function `void buildGraph(Graph *g)`, is the one implemented in Tutorial 004.

In the sample of code above, we create a graph with 3 properties : firstSelection (select), fisrtColors and firstMetric. We then set all nodes and edges "firstSelection" associated value to true which means that all nodes and edges are selected.

We, then, create two subgraphs out of our selection (the entire graph) :

```
//Create a hierarchy of subgraph (they all own the same elements)
Graph *subgraph1=graph->addSubGraph(select);
Graph *subgraph2=subgraph1->addSubGraph(select);
```

And, to finish this section, we add some new properties to those two subgraphs :

```
 //create a property in subgraph1 (redefinition of the one defined in graph)

subgraph1->getLocalProperty<DoubleProperty>("firstMetric");

//create a new property in subgraph1
subgraph1->getLocalProperty<DoubleProperty>("secondMetric");

//create a new property in subgraph3
subgraph2->getLocalProperty<DoubleProperty>("thirdMetric");
```

The property "firstMetric" will be redefined but not the two other ones.

3.11.2. 2. Properties of subgraph1

A good way to see what we have created is to iterate over the local properties of subgraph1 and in a second time iterate over inherited properties.

Following is a sample and its output that enables the iteration over local properties :

```
cout << "List of the local properties present in the subgraph1:" << endl;

Iterator<string> *it=subgraph1->getLocalProperties();
while (it->hasNext()) {
    cout << it->next() << endl;
  } delete it;
```

```
[root@atlas tutorial-005]# ./tutorial
List of the local properties present in the subgraph1:
firstMetric
secondMetric
```

As you can see the only local properties that has subgraph1 are "firstMetric" and "secondMetric". Indeed, "firstMetric" has been redefined, and, "thirdMetric" has been created with subgraph2.

Following is a sample and its output that enables the iteration over inherited properties :

```
  cout << "List of the inherited properties present in the subgraph1:" << endl;

  it=subgraph1->getInheritedProperties();
  while (it->hasNext()) {
    cout << it->next() << endl;
  } delete it;
```

```
 List of the local properties present in the subgraph1:
   firstColors
   firstSelection
```

As you can see, subgraph1 just has two inherited properties since "firstMetric" has been redefined.

Following is a sample of code that lists all the properties of a graph, the inherited properties and local properties :

```
  cout << "List of properties present in the subgraph1:" << endl;
  it=subgraph1->getProperties();
  while (it->hasNext()) {
    cout << it->next() << endl;
  } delete it;
```

```
List of properties present in the subgraph1:
firstMetric
secondMetric
firstColors
firstSelection
```

3.11.3. 3. Properties of subgraph2

As we did with subgraph1, we will now iterate over the local properties of subgraph2 in a first time and in a second time iterate over its inherited properties.

Following is a sample and its output that enables the iteration over local properties :

```
  cout << "List of the local properties present in the subgraph2:" << endl;

  it=subgraph2->getLocalProperties();
  while (it->hasNext()) {
    cout << it->next() << endl;
   } delete it;
```

```
[root@atlas tutorial-005]# ./tutorial
 List of the local properties present in the subgraph2:
 thirdMetric
```

The only local properties that has subgraph1 is thirdMetric.

Following is a sample and its output that enables the iteration over inherited properties :

```
  cout << "List of the inherited properties present in the subgraph2:" << endl;

  it=subgraph2->getInheritedProperties();
  while (it->hasNext()) {
    cout << it->next() << endl;
  } delete it;
```

```
List of the local properties present in the subgraph2:
firstColors
firstMetric
firstSelection
secondMetric
```

As you can see, subgraph2 has a lot of inherited properties since he is the subgraph of subgraph1 which is the subgraph of the root graph.

3.11.4. Source Code

```
#include <iostream>
#include <tulip/BooleanProperty.h>
#include <tulip/ColorProperty.h>
#include <tulip/DoubleProperty.h>

/**
```

```
 * Tutorial 005
 *
 * Create a graph hierarchy with several properties
 * Display the inherited and local properties in each graph
 */

using namespace std;
using namespace tlp;

void buildGraph(Graph *graph) {
 //add three nodes
 node n1=graph->addNode();
 node n2=graph->addNode();
 node n3=graph->addNode();
 //add three edges
 graph->addEdge(n2,n3);
 graph->addEdge(n1,n2);
 graph->addEdge(n3,n1);
}

int main() {
 //create an empty graph
 Graph *graph=tlp::newGraph();

 //build the graph
 buildGraph(graph);

 //Get and create several properties
 BooleanProperty *select=graph->getLocalProperty<BooleanProperty>("firstSelecti

 graph->getLocalProperty<ColorProperty>("firstColors");
 graph->getLocalProperty<DoubleProperty>("firstMetric");

 //init the selection in order to use it for building clone subgraph
 select->setAllNodeValue(true);
 select->setAllEdgeValue(true);

 //Create a hierarchy of subgraph (there are all the same)
 Graph *subgraph1=graph->addSubGraph(select);
 Graph *subgraph2=subgraph1->addSubGraph(select);

 //create a property in subgraph1 (redefinition of the one defined in graph)

 subgraph1->getLocalProperty<DoubleProperty>("firstMetric");

 //create a new property in subgraph1
 subgraph1->getLocalProperty<DoubleProperty>("secondMetric");

 //create a new property in subgraph3
 subgraph2->getLocalProperty<DoubleProperty>("thirdMetric");

 cout << "List of the local properties present in the subgraph1:" << endl;

 Iterator<string> *it=subgraph1->getLocalProperties();
 while (it->hasNext()) {
  cout << it->next() << endl;
 } delete it;
```

```
  cout << "List of inherited properties present in the subgraph1:" << endl;

  it=subgraph1->getInheritedProperties();
  while (it->hasNext()) {
    cout << it->next() << endl;
  } delete it;


  cout << "List of properties present in the subgraph1:" << endl;
  it=subgraph1->getProperties();
  while (it->hasNext()) {
    cout << it->next() << endl;
  } delete it;


  cout << "List of the local properties present in the subgraph2:" << endl;

  it=subgraph2->getLocalProperties();
  while (it->hasNext()) {
    cout << it->next() << endl;
  } delete it;

  cout << "List of inherited properties present in the subgraph2:" << endl;

  it=subgraph2->getInheritedProperties();
  while (it->hasNext()) {
    cout << it->next() << endl;
  } delete it;

  delete graph;
  return EXIT_SUCCESS;
}
```

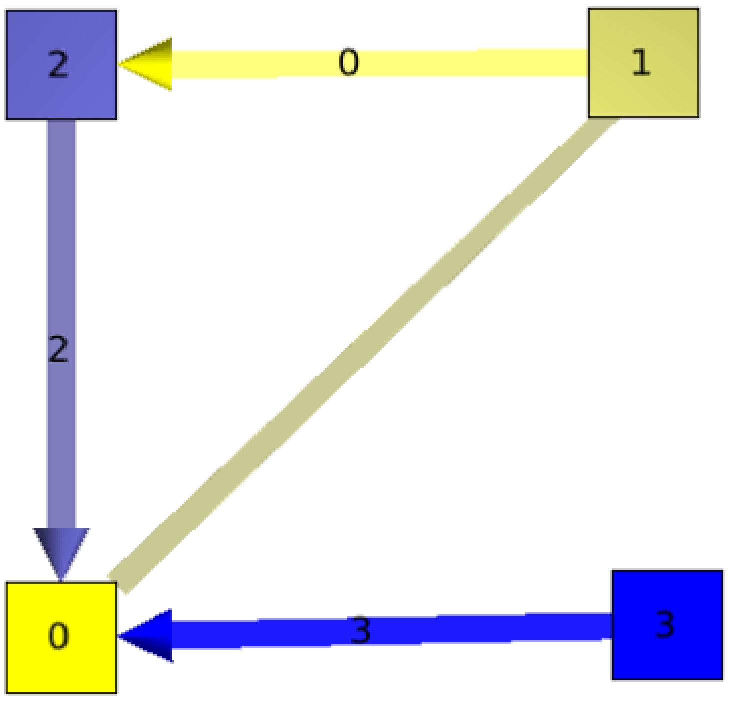
3.12. TUTORIAL 006 : Edges order.

In this tutorial, we will learn how to change edges order in the graph edges adjacency list (please visit Wikipedia: Adjacency and degree[5] for more details). Indeed, it can be useful to sort the edges considering a metric.

3.12.1. 1. Creation of the graph and its edges

We will create a graph with 4 nodes and 4 edges. Their "id number" will start from 0 just like in the figure below :

[5] http://en.wikipedia.org/wiki/Acyclic_Graph#Adjacency_and_degree



Following is the sample of code that created such a graph:

```
int main() {
  //create an empty graph
  Graph *graph=tlp::newGraph();

  //build the graph
  node n0=graph->addNode();
  node n1=graph->addNode();
  node n2=graph->addNode();
  node n3=graph->addNode();

  //add three edges
  edge e0=graph->addEdge(n1,n2);
  edge e1=graph->addEdge(n0,n1);
  edge e2=graph->addEdge(n2,n0);
  edge e3=graph->addEdge(n3,n0);
```

As you can see, node 0 has three edges : edge 1,edge 2 and edge 3. And if we display its edges adjacency list (see last section for function `void displayAdjacency(node n, Graph *graph)`) we obtain the following output :

```
 1 2 3
```

3.12.2. 2. Swap edges

Swapping edges can be easily done with the function, `void Graph::swapEdgeOrder ( const node, const edge,const  edge)` that will, as said swap two edges in the adjacent list of a node. Following is an example of its use :

```
  //swap e1 and e3
  graph->swapEdgeOrder(n0, e1, e3);
```

As you can see, the adjacency list has changed :

```
  3 2 1
```

3.12.3. 3. Setting an order

An other way to change the edges order is to use a vector of type edge and the function : `void Graph::setEdgeOrder (const node, const std::vector < edge > )`, following is an example that will replace e1 and e3 in their original order :

```
  vector<edge> tmp(2);
  tmp[0]=e1;
  tmp[1]=e3;
  graph->setEdgeOrder(n0,tmp);
```

And the new order :

```
  1 2 3
```

3.12.4. Source Code

```
#include <iostream>
#include <tulip/Graph.h>
#include <tulip/MutableContainer.h>

/**
 * Tutorial 006
 *
 * Create a graph
 * Order the edges around the nodes
 */

using namespace std;
using namespace tlp;

void buildGraph(Graph *graph) {
  //add three nodes
  node n0=graph->addNode();
  node n1=graph->addNode();
  node n2=graph->addNode();
  //add three edges
  graph->addEdge(n1,n2);
  graph->addEdge(n0,n1);
  graph->addEdge(n2,n0);
}

void displayAdjacency(node n, Graph *graph) {
  Iterator<edge>*ite=graph->getInOutEdges(n);
  while(ite->hasNext())
    cout << ite->next().id << " ";
  delete ite;
  cout << endl;
}

int main() {
  //create an empty graph
  Graph *graph=tlp::newGraph();

  //build the graph
  node n0=graph->addNode();
  node n1=graph->addNode();
  node n2=graph->addNode();
  node n3=graph->addNode();

  //add three edges
  edge e0=graph->addEdge(n1,n2);
  edge e1=graph->addEdge(n0,n1);
  edge e2=graph->addEdge(n2,n0);
  edge e3=graph->addEdge(n3,n0);

  tlp::saveGraph(graph, "adj1.tlp");

  //display current order of edge around n1
  displayAdjacency(n0,graph);

  //swap e1 and e3
  graph->swapEdgeOrder(n0,e1,e3);


  //display the new order of edge around n1
  displayAdjacency(n0,graph);

  vector<edge> tmp(2);
  tmp[0]=e1;
  tmp[1]=e3;
  graph->setEdgeOrder(n0,tmp);
  //display the new order of edge around n1
  displayAdjacency(n0,graph);
  delete graph;
  return EXIT_SUCCESS;
}
```

3.13. TUTORIAL 007 : Mutable Collection

In this small tutorial, we will learn how to use the Mutable Container (an efficient associative container) of the tulip API that enables :

- A tradeoff between speed and memory.
- To manage fragmented index

The direct access in this container is forbidden, but it exist a getter and a setter :

- `const ReturnType<TYPE >::ConstValue MutableContainer<type>::get(const unsigned int i) const` that returns a reference instead of a copy in order to minimize the number copy of objects, user must be aware that calling the set function can devalidate this reference

- `void MutableContainer<type>::set( const unsigned int i,const TYPE value).`

The MutableContainer has two more methods :

- `void setAll (const TYPE value)`

- `IteratorValue* findAll(const TYPE &value, bool equal = true) const`

Following is a small example of its use :

```
//declaration of a new MutableContainer
MutableContainer<int> t;

//set all element to 0
t.setAll(0);
//except element of index 1 set to 1.
t.set(1,1);

//display on standard output
cout << t.get(1) << "and" << t.get(2) << endl;
```

3.14. TUTORIAL 008 : Graph Tests

The tulip API has special functions to test if a graph is element of a specific class of graph. Moreover results are buffered and automatically updated if it is possible in constant time.

Specific functions are available for each test.

Each class of graph has its own header file. Following is a list of those header files and an example of their use :

- tulip/AcyclicTest.h `AcyclicTest::isAcyclic(graph);`

- tulip/BiconnectedTest.h `BiconnectedTest::isBiconnected(graph);`

- tulip/ConnectedTest.h `ConnectedTest::isConnected(graph);`

- tulip/OuterPlanarTest.h `PlanarityTest::isOuterPlanar(graph);`

- tulip/PlanarityTest.h `PlanarityTest::isPlanar(graph);`

- tulip/SimpleTest.h `SimpleTest::isSimple(graph);`

- tulip/TreeTest.h `TreeTest::isTree(graph);`

- tulip/TriconnectedTest.h `TriconnectedTest::isTriconnected(graph);`

3.15. TUTORIAL 009 : ObservableGraph

In this tutorial, we will show you how to use the class ObservableGraph that enables to receive notification when a graph is updated.

First, we will create a class that inherits from ObservableGraph, and then use it and see what this new class is able to "do".

3.15.1. 1. Our new class, GraphObserverTest :

Header files and namespaces :

To continue this tutorial, we need to include the following files, and namespaces.

```
#include <iostream>
#include <set>
//- - - - - - - - - - - - - -
#include <tulip/TlpTools.h>
#include <tulip/Graph.h>
#include <tulip/ObservableGraph.h>
//- - - - - - - - - - - - - -

using namespace std;
using namespace tlp;
```

As said before, we will create a new class that inherits from ObservableGraph. This class will have several functions to be notified when a node or an edge is deleted or added or reversed (edge).

Following is the class GraphObserverTest :

```
class GraphObserverTest : public GraphObserver {
public:
  GraphObserverTest() {
  }

private:
  void addEdge(Graph * g,const edge e) {
    cout << "edge :" << e.id << " has been added" << endl;
  }
  void delEdge(Graph *,const edge e) {
    cout << "edge :" << e.id << " is about to be deleted" << endl;
  }
  void reverseEdge(Graph *,const edge e) {
    cout << "edge :" << e.id << " is about to be reversed" << endl;
  }
  void addNode(Graph * g,const node n) {
    cout << "node :" << n.id << " has been added" << endl;
  }
  void delNode(Graph *,const node n) {
    cout << "node :" << n.id << " is about to be deleted" << endl;
  }
  void destroy(Graph *g) {
    cout << "graph : " << g->getId() << " is about to be deleted" << endl;

  }
};
```

Those methods are redefinitions of their homologues in the super class.

3.15.2. 2. The Main function

Following in the main function with some explanations :

```
int main(int argc, char **argv) {
  Graph *graph = tlp::newGraph();
```

Nothing really new here, just the creation of the graph.

```
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  GraphObserverTest graphObserverTest;
  graph->addObserver(&graphObserverTest);
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

We add the instance of our class GraphObserverTest with the function `Graph::addObserver` .

We then perform some modifications on the graph :

```
  node n1 = graph->addNode();
  node n0 = graph->addNode();
  edge e0  = graph->addEdge(n0, n1);
  graph->reverse(e0);
  graph->delNode(n0);
  delete graph;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  return EXIT_SUCCESS;
}
```

Following is the output of this program :

```
node :0 has been added
node :1 has been added
edge :0 has been added
edge :0 is about to be reversed
node :0 is about to be deleted
edge :0 is about to be deleted
```

Chapter 4. Tulip Open GL Library

4.1. Introduction

Tulip-ogl is a library for OpenGL allowing the developper to add augmented displays on the graph

4.2. 2D/3D for Tulip

4.2.1. Augmented Displays with GlEntity system

The objective of tulip-ogl's class is to be as modular as possible, giving the user a full scalable and Tulip-compatible engine to render custom augmented displays.

4.2.2. GlSimpleEntity

GlSimpleEntity is the mother-class of every 2D/3D shape. It provides one important virtual functions : draw, and one important data field : boundingBox. By generalizing this class, you can have classes making calls to OpenGL functions within the draw function. The other classes of the library are : GlLine, GlPolygon, GlBox, GlCircle, GlGrid, GlMultiPolygon, GlQuad, GlRect, GlRectTextured, GlSphere.

4.2.3. Gl 2D/3D classes

There are 4 basic Gl classes which display in 2D or 3D. Each of them represents a different primitive.

- *GlLine* : Describes a line according to a start position, a start color, an end position and an end color.
- *GlQuad* : Describes a quad according to a position and a size, or according to four points.
- *GlRect* : Describes a rectangle according to a position and a size, or according to two points.
- *GlPolygon* : Describes a polygon according to a vector of positions.
- *GlBox* : Displays a box. This augmented display is built with 6 GlQuad.
- *GlGrid* : Displays a 3D or a 2D projection of a grid. This augmented display is built with GlLine.
- *GlCircle* : Describes a circle according to a position and a size.
- *GlMultiPolygon* : Describes a set of polygon.
- *GlComplexPolygon* : Describe a convex or concav polygon with color texture and hole
- *GlRectTextured* : Describes a textured rectangle according to a position and a size, or according to two points.
- *GlSphere* : Describes a sphere according to a position and a size.

4.2.4. Scene and layers

The core of a view is the scene. A scene contain all opengl information and hierarchy of GlEntity to display graph and augmented displays

In the scene you have one or N layers

A layer contain a camera and a hierarchy of GlEntity

In NodeLinkDiagramComponent you have three layers : background (2D layer), Main (graph layer) and Foreground (2D layer).

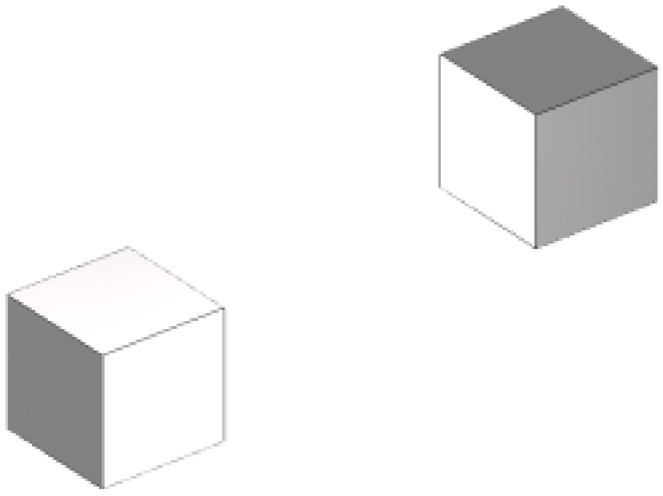
In following example you construct GlEntity and add it in the Main layer. To get this main layer you just need to call `scene->getLayer("Main")`

4.2.5. Examples of Gl shape uses

This section contains three small examples of use of the augmented displays with tulip. We admit the user has a GlLayer already defined in the variable *glMainLayer*.

In the given screenshots, the scene is composed of two nodes placed at positions (-1, -1, -1) and (1, 1, 1). Their sizes are (1, 1, 1)

Here is the base screenshot of the scene :



You can download plugins skeletons here[1] .

4.2.5.1. Example of Basic Gl 3D uses : GlLine

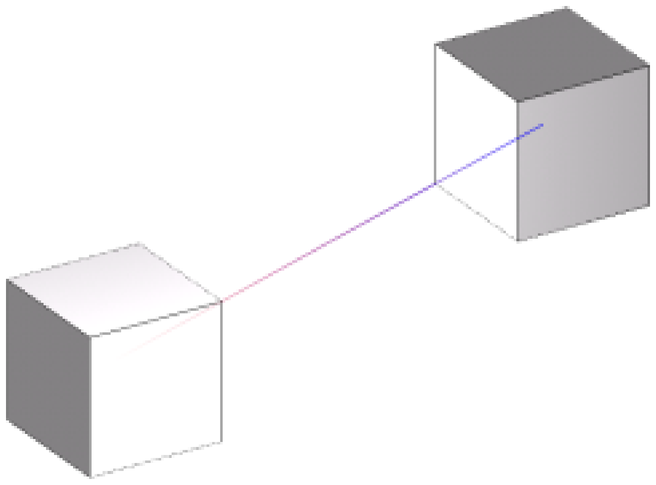
This simple example shows how to add a Line from the position (-1, -1, -1) to the position (1, 1, 1) as an augmented display in a GlScene, the line will be starting Blue and will finish transparent Red; the thickness of the line will be set to 1 pixel.

```
Coord startPos(-1, -1, -1);
Coord endPos(1, 1, 1);
Color startCol(0, 0, 255, 255);
Color endCol(255, 0, 0, 0);

// We create the line, the last parameter is the thickness of the line
 (note : you can't exceed 10)
```

[1] http://tulip.labri.fr/samples/plugintemplates.tar.gz

```
GlLine* line = new GlLine(startPos, endPos, startCol, endCol, 1);

// Finally we add the line to the GlLayer, naming it "The famous
 tutorial line"
glMainLayer->addGlEntity(line, "The famous tutorial Line");
```

Here is the screenshow of the result :

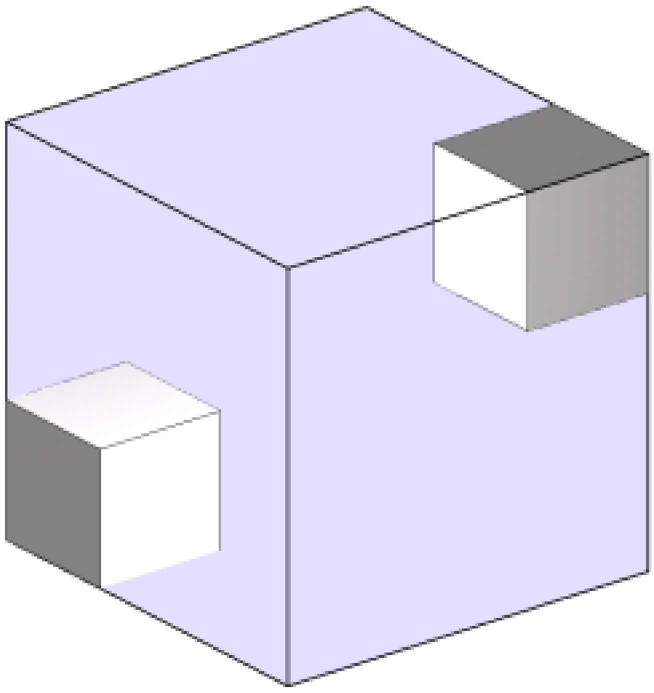


4.2.5.2. Example of Advanced Gl 3D uses : GlBox

This example shows how to add a Box from the position (-1.5, -1.5, -1.5) to the position (1.5, 1.5, 1.5) as an augmented display in a GlLayer. The box will be blue and semi transparent (220, 220, 255, 80).

```
Coord topLeft(-1.5, -1.5, -1.5);
Coord bottomRight(1.5, 1.5, 1.5);
Color boxColor(220, 220, 255, 80);

// We create the box by giving the bounding box to the constructor
// (topLeft and bottomRight) and the color of the box.
GlBox *box = new GlBox(topLeft, bottomRight, boxColor);

// Finally we add the box to the GlLayer, naming it
"Gl Tutorial 2 : Box"
glMainLayer->addGlEntity(box, "Gl Tutorial 2 : Box");
```

Here is the screenshot of the result :

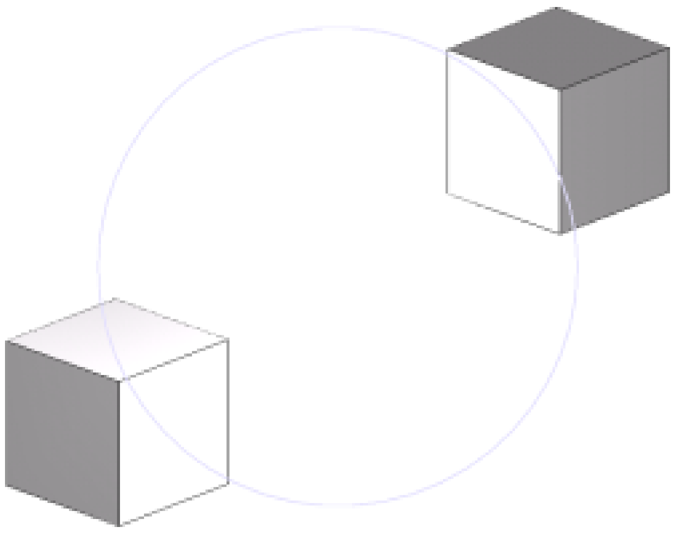


4.2.5.3. Example of Gl 2D uses : GlCircle

This example shows how to add a Circle centered at the middle of the screen, of a radius of 256 pixels(The screenshot has been scaled). The circle will be light blue and will have 50 segments

```
//We firstly add a new GlLayer (with name "2D layer") in the scene and set layer to 2

GlLayer *layer2D=new GlLayer("2D layer");
layer2D->set2DMode();
glScene->addLayer(layer2D);

//after, get the viewport to guess the center of the window
Vector<int, 4> viewport;
viewport = glScene->getViewport();

// This is the position of the center of the circle
// (ScreenWidth / 2, ScreenHeight / 2, 1)
Coord circleCenter(viewport[2] / 2, viewport[3] / 2, 0);
Color circleColor(220, 220, 255, 255);

// We create the circle giving it's center position, it's color,
 it's radius and the number of segments
GlCircle* circle = new GlCircle(circleCenter, circleColor,
 256, 50);

// Finally we add the circle to the GlLayer.
layer2D->addGlEntity(circle, "Gl Tutorial 3 : Circle");
```

Here is the screenshot of the result (the screenshot has been shrinked so the radius is not of 256 pixels) :



4.2.5.4. Compositing with Gl shape

This example shows how to use GlComposite to compose multiple effects in a scene. The scene will be composed of a sphere and 4 rectangles, to simulate an ArcBall

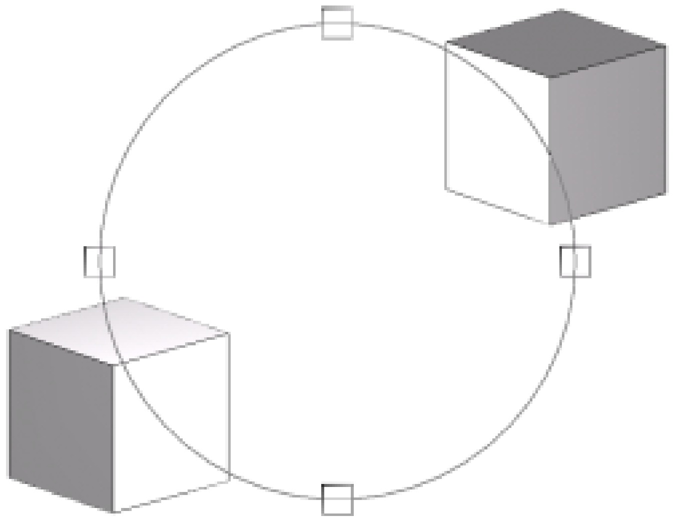
The circle will be positionned at the center of the screen. It will have a radius of 256 pixels and will be of a medium grey (128, 128, 128, 255).

The four squares will be positionned every 90° on the circle. They will also be in medium gray, and only wired

```
//We firstly add a new GlLayer (with name "2D layer") in the scene and set layer to 2

GlLayer *layer2D=new GlLayer("2D layer");
layer2D->set2DMode();
glScene->addLayer(layer2D);

// Create a new composite to store the final Augmented display :
GlComposite *composite = new GlComposite();

// This is the medium grey color that will be applied to every Gl shape :

Color color(128, 128, 128, 255);
// We get the viewport for the circle :
Vector<int, 4> viewport;
viewport = glScene->getViewport();

// This is the position of the center of the circle
// (ScreenWidth / 2, ScreenHeight / 2, 1)
Coord circleCenter(viewport[2] / 2, viewport[3] / 2, 0);

// We create the circle. It still have 50 segments
GlCircle* circle = new GlCircle(circleCenter, color, 256, 50);

// A 4 entries table for the squares
GlRect* rects[4];

Coord center, topLeft, bottomRight;
```

```
for(int i=0; i < 4; i++)
  {
   // We calculate the position of the center of each square
   center[0] = cos((double)i * 3.14/2.0) * 256;
   center[1] = sin((double)i * 3.14/2.0) * 256;
   center[2] = 0;
   center = center + circleCenter;

   // Then we find the position of the topLeft and the bottomRight corner

   topLeft    = center - Coord(16, 16, 0);
   bottomRight = center + Coord(16, 16, 0);

   rects[i] = new GlRect(bottomRight, topLeft, color, color);
  }

// We add the circle and the 4 squares to the composite
composite->addGlEntity(rects[0], "Gl Tutorial 4 : Rect1");
composite->addGlEntity(rects[1], "Gl Tutorial 4 : Rect2");
composite->addGlEntity(rects[2], "Gl Tutorial 4 : Rect3");
composite->addGlEntity(rects[3], "Gl Tutorial 4 : Rect4");
composite->addGlEntity(circle, "Gl Tutorial 4 : Circle");

// Finally we add the composite to the GlLayer
layer2D->addGlEntity(composite, "Composite");
```

Here is a screenshot of the result :



This is very simple but you can do more simply : a GlLayer is composed with a GlComposite, so you can add rectangles and circle directly to the layer2D, like this :

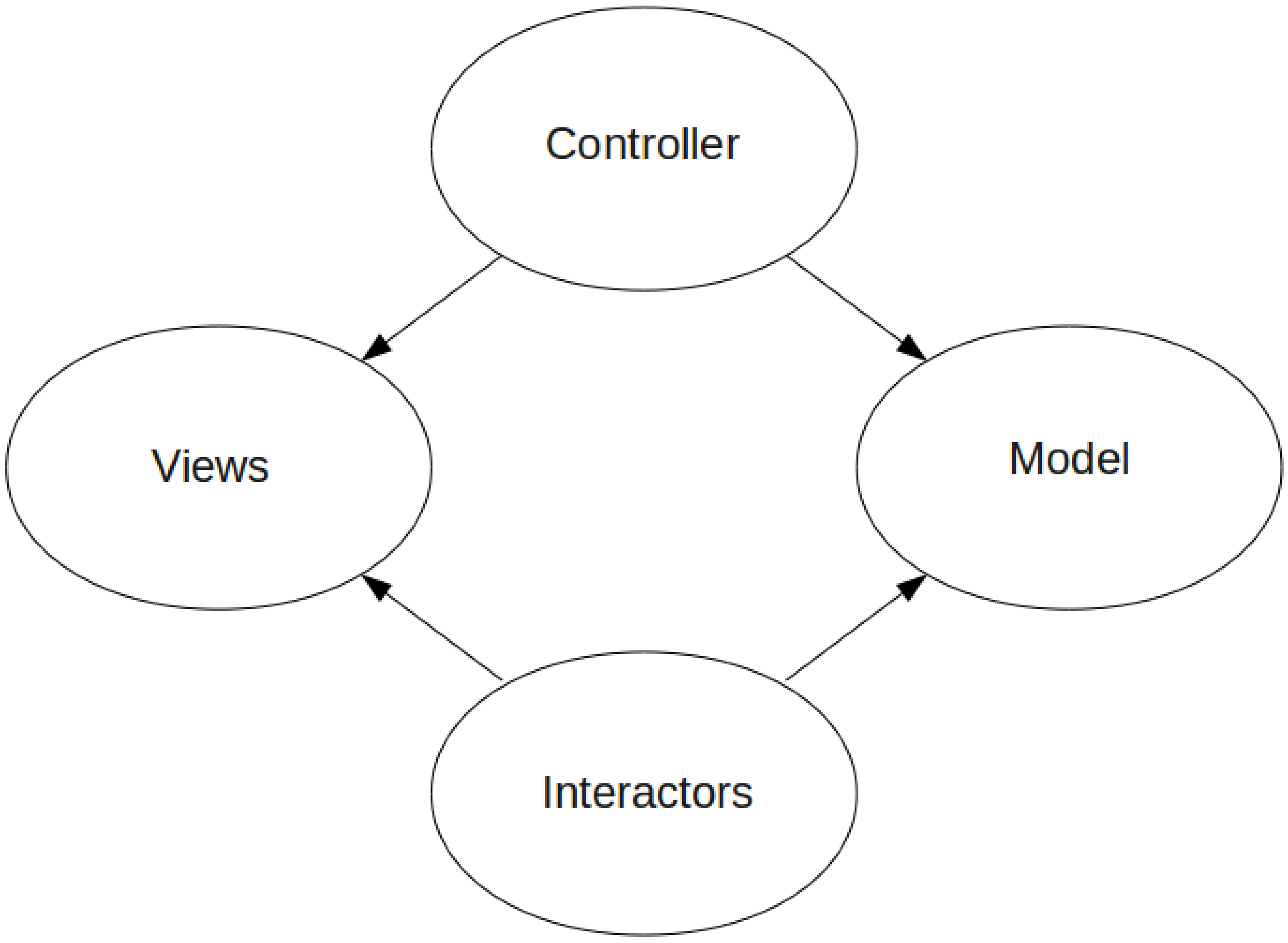
```
// We add the circle and the 4 squares to the GlLayer
layer2D->addGlEntity(rects[0], "Gl Tutorial 4 : Rect1");
layer2D->addGlEntity(rects[1], "Gl Tutorial 4 : Rect2");
layer2D->addGlEntity(rects[2], "Gl Tutorial 4 : Rect3");
layer2D->addGlEntity(rects[3], "Gl Tutorial 4 : Rect4");
layer2D->addGlEntity(circle, "Gl Tutorial 4 : Circle");
```

Chapter 5. Tulip QT Library

5.1. Introduction

In Tulip the plugin system is extended to interactors, views and controllers

- An interactor provide a mechanisme to modify view and/or data (graph)
- A view is a way to visualize graph (and others data if you want)
- A controller is here to change the aspect of Tulip and to manage views



Here you have a picture represent a modified Model-View-Controller architectural pattern

- Controller manage views and model (graph)
- Views use model to display it
- Interactors is a mini controller system who manage attached view and model

5.2. Interactors

An interactor is construct to interact with a view

In node link diagram view we have many interactors, for example : node builder interactor, selection interactor ... All these interactors have icons and this icons are visible in the toolbar

In interactor Tulip system we have two main classes : InteractorComponent and Interactor itself

Interactor is assembling a set of several interactor component

5.2.1. Interactor example

If you want, you can download an interactor example here[1]

Extract archive, go in the directory, modify your PATH environment varialbe, run make and make install

PATH environment variable must contain the directory where you install you tulip. Here you have an example to modify this variable : export PATH=/home/user/install/tulip/bin:$PATH

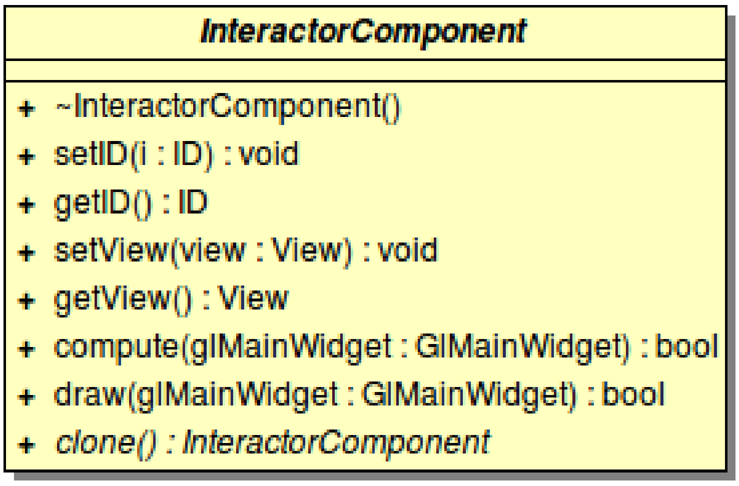
All the code of this interactor is commented inside

5.2.2. Interactor Component

An interactor component is the basic building block of the interactor system

To construct an interactor component you have to build a class that inherits InteractorComponent class

5.2.2.1. InteractorComponent class



InteractorComponent
+ ~InteractorComponent()
+ setID(i : ID) : void
+ getID() : ID
+ setView(view : View) : void
+ getView() : View
+ compute(glMainWidget : GlMainWidget) : bool
+ draw(glMainWidget : GlMainWidget) : bool
+ *clone() : InteractorComponent*

In InteractorComponent class we have 7 functions, but 4 of them are implemented on InteractorComponent. So if you want to create a new InteractorComponent, you have to implement 3 functions (4 if you count the eventFilter function)

- compute : This function is call before the rendering of the scene. It can be used to add GlEntities in the scene

[1] http://tulip.labri.fr/samples/interactorpluginexample.tar.gz

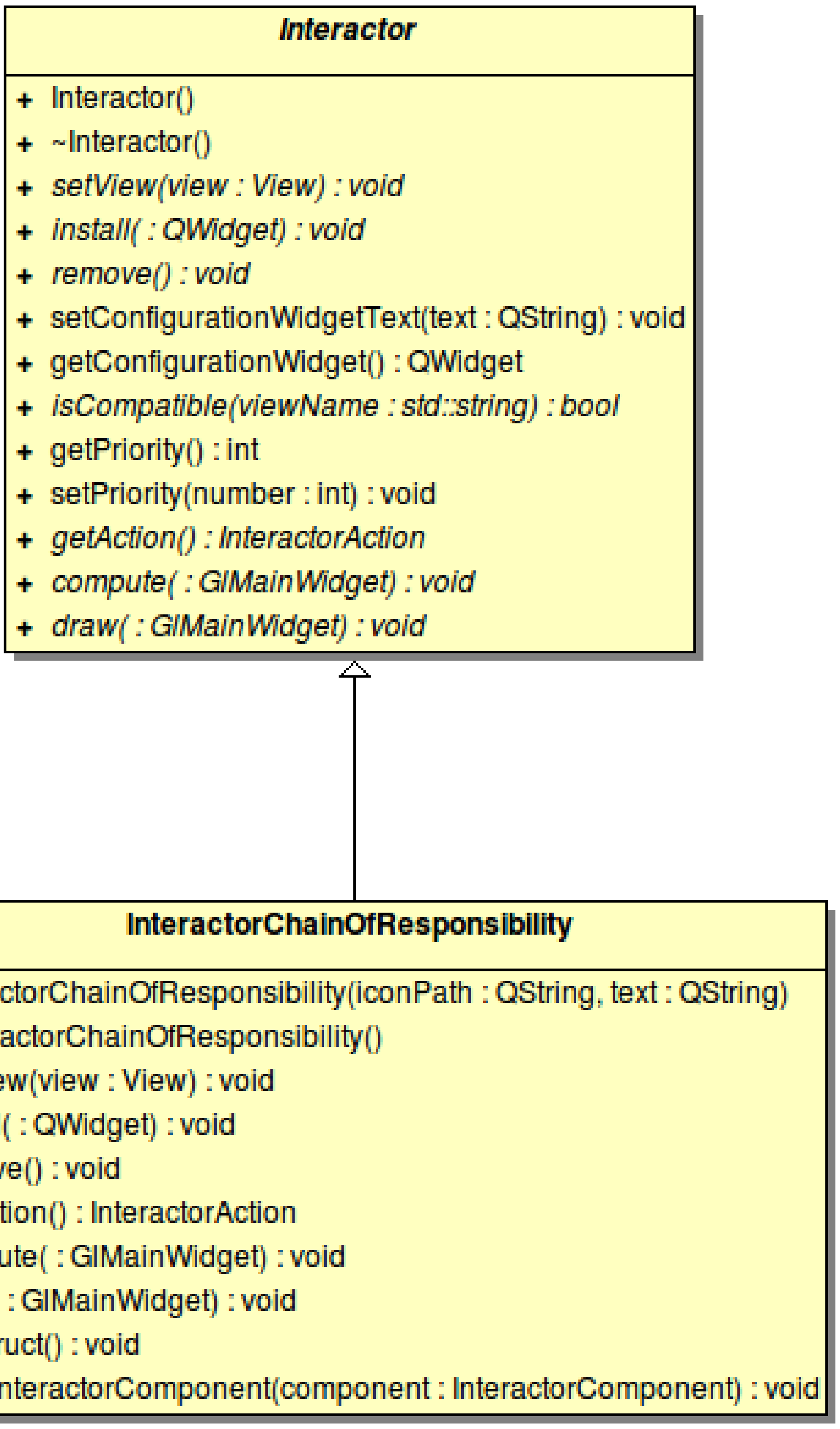
- draw : This function is call just after the graph rendering. It can be used to add some OpenGL object directly on the OpenGL viewFor example : selection interactor use this function to draw the selection rectangle
- clone : This function must be implemented to return a copy of the interactor component
- eventFilter : More important function ! This function is call by Qt to treat the event. You must reimplement this function and treat the event if you can

5.2.3. Interactor itself

You have two options to create an interactor :

- Directly implement Interactor interface
- Implement InteractorChainOfResponsibility class

InteractorChainOfResponsibility class is a first partial implementation of Interactor class with chain of responsibility system


Interactor
+ Interactor()
+ ~Interactor()
+ setView(view : View) : void
+ install(: QWidget) : void
+ remove() : void
+ setConfigurationWidgetText(text : QString) : void
+ getConfigurationWidget() : QWidget
+ isCompatible(viewName : std::string) : bool
+ getPriority() : int
+ setPriority(number : int) : void
+ getAction() : InteractorAction
+ compute(: GlMainWidget) : void
+ draw(: GlMainWidget) : void
InteractorChainOfResponsibility
+ InteractorChainOfResponsibility(iconPath : QString, text : QString)
+ ~InteractorChainOfResponsibility()
+ setView(view : View) : void
+ install(: QWidget) : void
+ remove() : void
+ getAction() : InteractorAction
+ compute(: GlMainWidget) : void
+ draw(: GlMainWidget) : void
+ construct() : void
# pushInteractorComponent(component : InteractorComponent) : void

5.2.3.1. InteractorChainOfResponsibility concept

In this class the interactor system is build with the chain of responsibility pattern

Here we have a chain of responsibility of InteractorComponent

You have a list of InteractorComponent. When Qt have an event, this event is passed to first element of this list, if InteractorComponent don't treat this event (function eventFilter return false) the event is passed to second element of the list ...

5.2.3.2. InteractorChainOfResponsibility class

If you want to use InteractorChainOfResponsibility system, you have some class to implement :

- In you constructor :

- Call InteractorChainOfResponsibility constructor with two string. First is the path to interactor icon, second is the interactor tooltip text.
- Call setPriority(int) function to set the priority of the interactor in the toolbar : if priority is high (>5 for example) interactor icon is placed at left side of the toolbar
- Call setConfigurationWidgetText("Text displayed in the configuration widget of the interactor"), an other solution is to reimplement the getConfigurationWidget function who return the QWidget used to configure this interactor

- Implement construct() function : in this function you have to call pushInteractorComponent(new nameOfYourInteractorComponent) to add interactor component in the chain of responsibility

- Implement isCompatible(string viewName) function, if your interactor is compatible with view (with name viewName) this function must return true, else return false

5.2.3.3. Interactor class

If you want to directly use Interactor class (if you doesn't want the chain of responsibility system), you have to implement function :

setView(View *), install(QWidget *), remove(), isCompatible(string), getAction(), compute() and draw()

And you have to call setConfigurationWidgetText(string) (or reimplement getConfigurationWidget()) and setPriority(int)

Interactor function documentation is available in the Library API page

5.2.4. INTERACTORPLUGIN macro

After inplementation of InteractorComponent and Interactor, you have to add you interactor to plugin system :

To do that you have to call INTERACTORPLUGIN macro

INTERACTORPLUGIN(InteractorPluginClassName, "InteractorPluginName", "Authors", "Date", "Interactor plugin full name", "plugin release");

5.3. Views

A view is used to display graph data

In Tulip base you have two different views : Node link diagram view and Spreadsheet view

5.3.1. View example

If you want, you can download an view example here[2]

Extract archive, go in the directory, modify your PATH environment variable, run make and make install

PATH environment variable must contain the directory where you install you tulip. Here you have an example to modify this variable : export PATH=/home/user/install/tulip/bin:$PATH

All the code of this view is commented inside

5.3.2. How to create a view

If you want to create a new view, you have two things to do :

5.3.2.1. Implement View interface

First you have to implement View interface

This interface can be separated into three parts :

- Data part : `QWidget *construct(QWidget *parent);`

```
// This function is call by the controller when it want to load this view
// This function must return a QWidget where we have the view
void setData(Graph *,DataSet);
// This function is called when we load a new graph or when we create a new
one
// In the DataSet you can have nothing when you create a new graph or you
can have old data when you load a tlp file
void getData(Graph **, DataSet *);
// This function is called when you save your graph
// You must store the graph and the view data
voif setGraph(graph *);
// This function is called when you change the visualized graph in
hierarchical view (for example)
Graph *getGraph();
```

- Interactors part : `void setInteractors(const std::list<Interactor *> &interactors);`

```
// This function is call by the controller to give interactors to view
std::list<Interactor *> getInteractors();
// This function must return interactors of this view (previously given
interactors)
void setActiveInteractor(Interactor *interactor);
// This function is call by the controller when an interactor is activate
on this view
```

[2] http://tulip.labri.fr/samples/viewpluginexample.tar.gz

• Display part : `void draw();`

```
// Call when the graph is modified or by interactors
void redraw();
// Mainly call by Qt when a menu is open in front of the view
// Data is not modified so the view is unchanged
void init();
// Call when the graph is modified and when the view need to be completely↵
init and draw
// For example this function is call when a layout algorithm is running, in↵
NodeLinkDiagramComponent this function call centerView() and draw()
void createPicture(const std::string &pictureName,int width=0, int↵
height=0);
// Create a picture of this view and store it in file with pictureName
```

5.3.2.2. Call View plugins macro

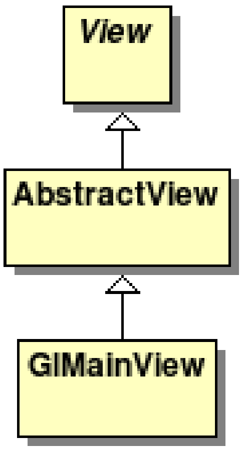
The second thing you have to do is register your view with the ViewManager

To do that, you just have to call VIEWPLUGIN macro

```
VIEWPLUGIN(ViewClassName, "ViewName", "Authors", "Date", "Small View description",
```

5.3.3. An other solution : Use AbstractView or GlMainView

If you doesn't want to implement all View function, you can implement a class who inherits of AbstractView class or GlMainView class



• View : is the main interface without implementation

• AbstractView : provide a basic implementation of interactor system

• GlMainView : provide a OpenGl view with overview

5.3.3.1. AbstractView class

AbstractView class provide a basic implementation of iteractor system, so if you want to use it you don't have to implement functions : setInteractors(...), getInteractors(...), setActiveInteractor(...) and getActiveInteractor(...)

But in your construct(..) function implementation, you have to call AbstractView::construct(...). In addition, AbstractView::contruct(...) function create a QWidget for the view, so you just have to write a code like this :

```
QWidget *YouViewClass::construct(QWidget *parent) {
  QWidget *widget=GlMainView::construct(parent);
  Qwidget *yourWidget=new QWidget;
  ...
  setCentralWidget(yourWidget);
  return widget;
}
```

In addition, AbstractView provide a system to create a context menu (the menu that appears when you click right mouse button in the view)

In this part you have 3 different functions :

- buildContextMenu function : here you construct your menu, to do that you add QMenu in contextMenu parameters, for example : `contextMenu->addMenu(new QMenu("viewMenu"));`
- computeContextMenu function : this function is call when the user click on a menu in context menu. In this function you have to threat this action
- Finaly you have specificEventFilter function : this function is call before all others function if the user move/click the mouse. You have to implement (if you want) your specific mouse mechanism on this function

5.3.3.2. GlMainView class

GlMainView class provide a OpenGl view with an overview

At default this OpenGl view display the Graph, but you can modify this default system. GlMainView use GlMainWidget system to display graph (see GlMainWidget section for more informations

In your view contruct(...) function you have to call GlMainView::construct(..)

This class implement init(...), draw(...), refresh(...) and createPicture(...) functions

In addition, this class add an hide overview button in contextMenu, but you can add your own menus in context menu (as you do with AbstractView)

5.3.3.2.1. GlMainView getData(..) and setData(...)

To implement your own methodes, you have two solution :

- First method is ok if you want to create a view who display the graph with node/link system, this system use setData and getData of GlMainWidget

- If you want you own OpenGl visualisation, you must rebuild view in setData and store data in getData

5.3.3.2.1.1. GlMainView with node/link visualisation system

For more simple explanation, we focus initially to getData(...) function

As we said above, the GlMainView use a GlMainWidget to display Graph (and other things if you want), so in getData you have to store GlMainWidget data and, if you want, you can store your own data

To do this you must have a code like this

```
void YourView::getData(Graph **graph,DataSet *dataSet) {
  dataSet->set<DataSet>("glMainWidgetData",mainWidget->getData());
  dataSet->set<string>("owndata","an example of own data");
  *graph=mainWidget->getGraph();
}
```

Is very simple

Now we focus on setData(...) function

This function is call by the controler to load (previously stored) data on your view

Like getData function, you have to load data in GlMainWidget class

You must have a code like this

```
void YourView::setData(Graph *graph, DataSet data){
  DataSet glMainWidgetData;
  string stringData;

  //We check if glMainWidgetData exist because getData can be call with empty data s

  if(data.exist("glMainWidgetData")
    data.get("glMainWidgetData",glMainWidgetData);
  if(data.exist("owndata")
    data.get("owndata",stringData);

  mainWidget->setData(graph,glMainWidgetData);
}
```

5.3.3.2.1.2. GlMainView with more complex visualisation

At this point you can't use getData and setData os glMainWidget

You have to completely reload you view in setData function

Why ? This is simple, setData of GlMainWidget is made to automaticaly create a node/link visualisation

So you have to manually load the data in GlMainWidget

For example if you want a view that show a simple sphere, you can create a code like this :

```
void YourView::setData(Graph *graph, DataSet data){
  this->graph=graph;

  //Create a layer with name "Main" is very important, because GlMainWidget use it

  GlLayer *mainLayer=new GlLayer("Main");
  glMainWidget->getScene()->addLayer(mainLayer);

  GlSphere *sphere;
  if(data.exist("spherePosition"){
    Coord position=data.get("spherePosition",position);
    sphere=new GlSphere(position,10.);
  }else{
    sphere=new GlSphere(Coord(0,0,0),10.);
  }

  mainLayer->addGlEntity("Sphere",sphere);
}
```

And if you want to store data of your view, you can do this :

```
void YourView::getData(Graph **graph,DataSet *data){
  *graph=this->graph;
  data->set<Coord>("spherePosition",spherePosition);
}
```

5.3.3.2.2. Overview system

At default, if you create your view, the overview will display anything because you have to give some informations to overview

You just have to call setObservedView(GlMainWidget *widget,GlSimpleEntity *entity) function

In the overview, you can display only one element (but this element can be a GlComposite) and this element must be store in entity parameter

So if you have you view which inherits of GlMainView class and you want to display the graph on the overview (graph is display at default in GlMainView), you have to enter a line like this in your setData(...) function

```
overviewWidget->setObservedView(mainWidget,mainWidget->getScene()->getGlGraphCom
```

5.4. Controllers

As the majority of features in eclipse, the main window's layout is controlled by plugins. The base layout is set-up by the MainController class which inherits from the Controller abstract object.

Whenever a graph is loaded into Tulip, a new controller is chosen to display it (which means that there is as many controller objects as graphs loaded into Tulip). By default, only MainController is present and is always chosen to display new graphs. However, if several controllers are available, the user will be able to chose between them when a new graph will be loaded.

First, we will create a very basic controller which do practically nothing. We'll then use it as a strat to develop a ControllerPluginExample (whose features are described below).

5.4.1. Empty Controller

The first step to create a controller is to build the skeleton that will be the common base between all the controller plugins you'll write.

We are going to write a controller that does practically nothing (we'll call it VoidController) and use it as a base to develop our BasicExampleController.

The process is pretty similar to other plugins. First, we'll make a class that inherits from the Controller interface, overriding some methods, then we'll declare our new class as a controller plugin using the CONTROLLERPLUGIN

5.4.1.1. VoidController.h

```
#ifndef VOIDCONTROLLER_H_
#define VOIDCONTROLLER_H_

#include <tulip/Controller.h>
#include <tulip/DataSet.h>

class VoidController: public tlp::Controller {
public:
   VoidController();
   virtual ~VoidController();

   virtual void attachMainWindow(tlp::MainWindowFacade facade);
   virtual void setData(tlp::Graph *graph=0,
                  tlp::DataSet dataSet=tlp::DataSet());
   virtual void getData(tlp::Graph **graph, tlp::DataSet *data);
   virtual tlp::Graph *getGraph();

private:
   tlp::Graph *graph;
};


#endif /* VOIDCONTROLLER_H_ */
```

5.4.1.2. VoidController.cpp

```
#include "VoidController.h"

using namespace tlp;
using namespace std;

CONTROLLERPLUGIN(VoidController, "VoidController", "Author","12/02/2009","Tutori
VoidController::VoidController() {
}

VoidController::~VoidController() {
}

void VoidController::attachMainWindow(tlp::MainWindowFacade facade) {

 Controller::attachMainWindow(facade);
}

void VoidController::setData(Graph *graph, DataSet dataSet) {
 this->graph = graph;
}

void VoidController::getData(Graph **graph, DataSet *data) {
 *graph = this->graph;
}

Graph *VoidController::getGraph() {
 return this->graph;
}
```

5.4.1.3. Description

The code itself is pretty self-explanatory but we'll review some of its elements.

First, we can recognize familiar methods: setData, getData and getGraph aim at the same thing as their counterparts in other plugins. Each controller is assosiated to one graph (passed via setData) and can store some restoration infos via getData and getGraph methods.

The attachMainWindow is used to build the controller's GUI. A basic main window layout is given (the facade parameter). It contains the "File", "Windows" and "Help" menus and the "Open", "Save" and "Print" toolbar buttons. You can tune-up this GUI by adding whatever elements in this tulip window.

Controllers are tabbed elements: because you have one controller per graph, if multiple controllers are simultaneously loaded into Tulip, the user we'll be able to choose between them to see every new opened graph.

5.4.2. Controller example

If you want, you can download an controller example here[3]

Extract archive, go in the directory, modify your PATH environment varialbe, run make and make install

PATH environment variable must contain the directory where you install you tulip. Here you have an example to modify this variable : export PATH=/home/user/install/tulip/bin:$PATH

All the code of this controller is commented inside

5.4.3. ControllerPluginExample

[3] http://tulip.labri.fr/samples/controllerpluginexample.tar.gz

Now we will create a very basic controller. This controller will be create a Node Link Driagram View in full window mode, and you will have interactors to modify the graph

5.4.3.1. ControllerPluginExample.h

```
#ifndef CONTROLLERPLUGINEXAMPLE_H
#define CONTROLLERPLUGINEXAMPLE_H

#include <tulip/ControllerViewsManager.h>
#include <tulip/DataSet.h>
#include <tulip/Graph.h>
#include <tulip/AbstractProperty.h>
#include <tulip/Observable.h>

// For this example we construct an simple controller who display a simple node link

// This Controller use implementation of ControllerViewsManager
// And observe graph and properties of graph
class ControllerPluginExample: public tlp::ControllerViewsManager, public tlp::Ob

public:

   virtual ~ControllerPluginExample();

   // This function is call when tulip want to open it
   virtual void attachMainWindow(tlp::MainWindowFacade facade);

   // This function is call when tulip want to attach data on this controller

   virtual void setData(tlp::Graph *graph=0,tlp::DataSet dataSet=tlp::DataSet());


   // This function is call when tulip want to save this controller
   virtual void getData(tlp::Graph **graph, tlp::DataSet *data);

   // Return the current graph of this controller
   virtual tlp::Graph *getGraph();

   // This function is call by Observable
   // In setData, we add this controller to graph observer and properties observer

   // So this function is call when the graph is modified or when a property is modi

   virtual void update( std::set< tlp::Observable * >::iterator begin, std::set< tl


   // This function is need by Oberver class
   virtual void observableDestroyed(tlp::Observable*){}

private:
   tlp::Graph *graph;
   tlp::View *nodeLinkView;
};

#endif
```

5.4.3.2. ControllerPluginExample.cpp

```
#include "ControllerPluginExample.h"

#include <tulip/View.h>

using namespace std;
using namespace tlp;


// VIEWPLUGIN(ClassName, "ControllerName", "Authors", "date", "Long controller plu

CONTROLLERPLUGIN(ControllerPluginExample, "ControllerPluginExample", "Author","1


ControllerPluginExample::~ControllerPluginExample() {
 // when we delete this controller, we remove it of observer
 if(graph){
 Iterator<PropertyInterface*> *it = graph->getObjectProperties();
    while (it->hasNext()) {
       PropertyInterface* tmp = it->next();
       tmp->removeObserver(this);
    } delete it;

 graph->removeObserver(this);
 }
}

void ControllerPluginExample::attachMainWindow(tlp::MainWindowFacade facade) {

 // Call the attachMainWindow of ControllerViewsManager
 ControllerViewsManager::attachMainWindow(facade);
}

void ControllerPluginExample::setData(Graph *graph, DataSet dataSet) {

 // When we setData, we create a Node link diagram view

nodeLinkView=ControllerViewsManager::createView("Node Link Diagram view",graph,da

 this->graph = graph;

 // We set observer to observe properties and graph
 Iterator<PropertyInterface*> *it = graph->getObjectProperties();
    while (it->hasNext()) {
       PropertyInterface* tmp = it->next();
       tmp->addObserver(this);
    } delete it;

 graph->addObserver(this);
}

void ControllerPluginExample::getData(Graph **graph, DataSet *data) {

 *graph = this->graph;
}
```

```
Graph *ControllerPluginExample::getGraph() {
 return this->graph;
}

void ControllerPluginExample::update ( std::set< tlp::Observable * >::iterator beg

 // When graph or property is modified, we draw the view
 nodeLinkView->draw();
}
```

5.4.3.3. Description

In this example, we have two important things

- First we have the ControllerViewsManager classThis class is a tools class to create and manage views. In this example, we just use it to simplify creation of the view and activation of this view (and interactor activation). If we had directly created the view, without ControllerViewsManager, the interactor tool bar was empty

- The second important things is the observer systemIn Tulip graph and properties can be observed, to do this you just have to create a class that inherits of Observer and you have to connect observables (graph and properties) to this observer with addObserver(...) functionBut you can inherit of two others class : GraphObserver and PropertyObserver. GraphObserver is use when you want to know when a node is add/delete for example (see ObservableGraph class for more information). And PropertyObserver is use if you want to know when a property is modified (see ObservableProperty class for more information)

5.4.4. ControllerAlgorithmTools class

This class is a set of static function for the application of algorithm and for the test of structural properties of the graph

All these functions can be separated into three groups :

- Algorithm application functions : in this group you have getPluginParameters(...), applyAlgorithm(...), changeProperty(...), changeString(...), changeBoolean(...), changeMetric(...), changeLayout(...), changeInt(...), changeColors(...) and changeSizes(...) functions

- Structural test functions : isAcyclic(...), isSimple(...), isConnected(...), isBiconnected(...), isTriconnected(...), isTree(...), isFreeTree(...), isPlanar(...) and isOuterPlanar(...)

- Structural modification functions : makeAcyclic(...), makeSimple(...), makeConnected(...), makeBiconnected(..) and makeDirected(...)

5.4.5. ControllerViewsTools class

In this class you have a set of static function to create and use views and iteractors

You have five functions :

- createView(..) to create a view

- createMainView(..) to create a node link diagram view

- installInteractors(...) to install interactors of a view in tool bar
- changeInteractor(...) to activate interactor when we click on its icon
- getNoInteractorConfigurationWidget() to have an empty configuration widget for interactor

5.4.6. ControllerViewsManager class

This class provide a basic system to manage a multi views system

To use it, you have to create a view that inherit of this class and call ControllerViewsManager::attachMainWindow(...) in your class attachMainWindow function

Functions of ControllerViewsManager can be separate in two big group :

- Setter and accessor :

- getCurrentGraph() and getCurrentView() : to get current active view and graph associated with this view
- getViewsNumber() : to get number of view
- getGraphOfView(...), setGraphOfView(...), getViewOfWidget(...), setViewOfWidget(...), getWidgetOfView(...), getNameOfView(...) and setNameOfView(...) : to get/set association

- Tool funstions :

- createView(...) : to create a view
- installInteractors(...) : to install interactors for a view (normaly you don't have to call manualy this function, she is atoucall when view is activated)
- updateViewsOfGraph(...) and updateViewsOfSubGraphs(...) to update views
- changeGraphOfViews(...) : just see name of function ;)
- drawViews(...) : to draw all views
- saveViewsGraphsHierarchies() and checkViewsGraphsHierarchy() : to save hierarchy of graph of views and to check if hierarchy is good (after subgraph removing for example)

Chapter 6. Plug-ins development

Tulip has been built to be easily extensible. Therefore a mechanism of plug-ins has been set up. It enables to directly add new functionalities into the Tulip kernel. One must keeps in mind that a plug-in have access to all the parts of Tulip. Thus, one must write plug-ins very carefully to prevent memory leak and also errors. A bug in plug-in can result in a "core dump" in the software that uses it. To enable the use of plug-ins, a program must call the initialization functions of the plug-ins. This function loads dynamically all the plug-ins and register them into a factory that will enable to directly access to it.

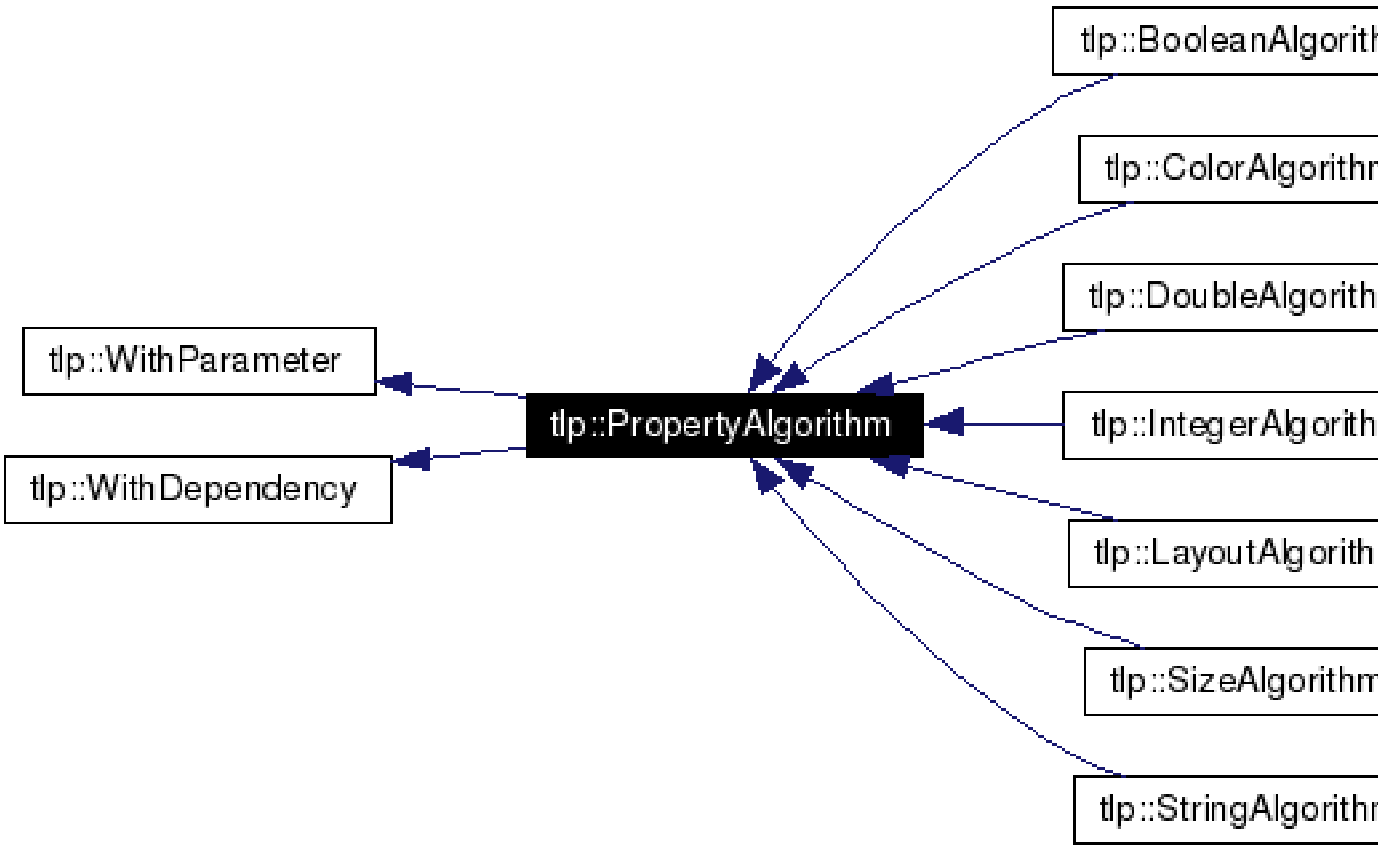
To develop a plug-in, you need to create a new class that will inherits from a specific algorithm class. Algorithms classes are separated in 4 different types :

Basic classes :

- `Property algorithms` : This kind of plug-ins will only affect a specific property of any type. See Section 6.1, “The PropertyAlgorithm class.”.
- `Generic algorithms` :This kind of plug-ins can be used to modify the entire graph. In other case it is preferable to use a specific class of algorithms to implement your plug-in. See Section 6.2, “The Algorithm class.”.
- `Importation algorithms` :Plug-ins to import a specific graph. For example a plug-in that will create a graph from a file directory system. See Section 6.3, “Import plug-ins”.
- `Export algorithms` : Plug-ins to save a graph to a specific type of file. See Section 6.4, “Export plug-ins”.

6.1. The PropertyAlgorithm class.

The PropertyAlgorithm class, is the class from which inherits different types of algorithms such as the BooleanAlgorithm class or the LayoutAlgorithm class (see picture below). This class is important in the way that every specific algorithm that you will develop will have to inherit from one of those classes. For example, if you write a plug-in to update the graph layout, your new class will have to inherit from the LayoutAlgorithm class which inherits from this PropertyAlgorithm class.



Each one of the 8 classes presented above has a public member, *TypeName*`Property`* *typeName*`Result`, which is the data member that which have to be updated by your plug-in. After a successful run tulip will automatically copy this data member into the corresponding property of the graph. Following is a table showing the data member, graph property and 'Algorithms' GUI submenu corresponding to each subclass of Algorithm :

Class name	**Data member**	**Graph property replaced**	**Algorithms GUI submenu**
BooleanAlgorithm	booleanResult	viewSelection	Selection
ColorAlgorithm	colorResult	viewColor	Color
DoubleAlgorithm	doubleResult	viewMetric	Measure
Algorithm	NA	NA	General
IntegerAlgorithm	integerResult	viewInt	Integer
LayoutAlgorithm	layoutResult	viewLayout	Layout
SizeAlgorithm	sizeResult	viewSize	Size
StringAlgorithm	stringResult	viewLabel	Label

Note that at anytime, if the user clicks on the "cancel" button (see Section 6.1.3, “The PluginProgress class.” for more details), none of your algorithm’s actions will changes the graph since the variable *typeName*Result is not copied in the corresponding property.

6.1.1. Overview of the class

A quick overview of the functions and data members of the class PropertyAlgorithm is needed in order to have a generic understanding of its 8 derived classes.

6.1.1.1. Public members

Following is a list of all public members :

- `PropertyAlgorithm (const PropertyContext& context)` :The constructor is the right place to declare the parameters needed by the algorithm. `addParameter<DoubleProperty>("metric", paramHelp[0], 0, false);` And to declare the algorithm dependencies.`addDependency<Algorithm>("Quotie Clustering", "1.0");`

- `~PropertyAlgorithm ()` : Destructor of the class.

- `bool run ()` : This is the main method : - It will be called out if the pre-condition method (bool check (..)) returned true. - It is the starting point of your algorithm. The returned value must be true if your algorithm succeeded.

- `bool check (std::string& errMsg)` : This method can be used to check what you need about topological properties of the graph, metric properties on graph elements or anything else.

6.1.1.2. Protected members

Following is a list of all protected members :

- `Graph* graph` :This graph is the one given in parameters, the one on which the algorithm will be applied.

- `PluginProgress* pluginProgress` : This instance of the class PluginProgress can be used to have an interaction between the user and our algorithm. See the next section for more details.

- `DataSet* dataSet` :This member contains all the parameters needed to run the algorithm. The class DataSet is a container which allows insertion of values of different types. The inserted data must have a copy-constructor well done. See the section called DataSet for more details.

The methods of the *TypeName*Algorithm class, will be redefined in your plug-in as shown in Section 6.1.4, “Example of a plugin skeleton”.

6.1.2. Parameters :

Your algorithm may need some parameters, for example a boolean or a property name, that must be filled in by the user just before being launched. In this section, we will look at the methods and techniques to do so.

6.1.2.1. Adding parameters to an algorithm

The class PropertyAlgorithm inherits from a class called WithParameters that has a member function named `template<typename Type> void addParameter (const char *name, const char *inHelp=0, const char *inDefValue=0, bool isMandatory=true)` which is capable of adding a parameter. This method has to be called in the constructor of your class.

Following is a description of its parameters :

- name : Name of the new parameter.
- inHelp : This parameter can be used to add a documentation to the parameter (See example below).
- inDefValue : Default value.
- isMandatory : If false, the user must give a value.

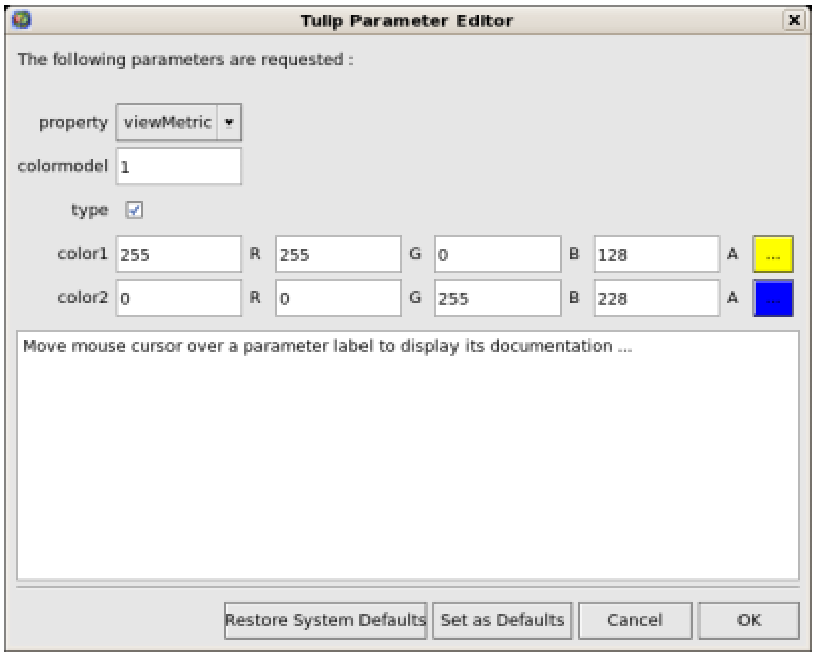
On the following example, we declare a character buffer that will contain the documentation of our parameters.

```
namespace {
  const char * paramHelp[] = {
    // property
    HTML_HELP_OPEN() \
    HTML_HELP_DEF( "type", "DoubleProperty" ) \
    HTML_HELP_BODY() \
    "This metric is used to affect scalar values to graph items." \
    "The meaning of theses values depends of the choosen color model." \

    HTML_HELP_CLOSE(),
    // colormodel
    HTML_HELP_OPEN() \
    HTML_HELP_DEF( "type", "int" ) \
    HTML_HELP_DEF( "values", "[0,1]" ) \
    HTML_HELP_DEF( "default", "0" ) \
    HTML_HELP_BODY() \
    "This value defines the type of color interpolation. Following values are valid

    "<ul><li>0: HSV interpolation ;</li><li>1: RGB interpolation</li></ul>" \

    HTML_HELP_CLOSE(),
    // color1
    HTML_HELP_OPEN() \
    HTML_HELP_DEF( "type", "Color" ) \
    HTML_HELP_DEF( "values", "[0,255]^4" ) \
    HTML_HELP_DEF( "default", "red" ) \
    HTML_HELP_BODY() \
    "This is the start color used in the interpolation process." \
    HTML_HELP_CLOSE(),
    // color2
    HTML_HELP_OPEN() \
    HTML_HELP_DEF( "type", "Color" ) \
    HTML_HELP_DEF( "values", "[0,255]^4" ) \
    HTML_HELP_DEF( "default", "green" ) \
    HTML_HELP_BODY() \
    "This is the end color used in the interpolation process." \
    HTML_HELP_CLOSE(),
    // Mapping type
    HTML_HELP_OPEN() \
    HTML_HELP_DEF( "type", "Boolean" ) \
    HTML_HELP_DEF( "values", "true / false" ) \
    HTML_HELP_DEF( "default", "true" ) \
    HTML_HELP_BODY() \
    "This value defines the type of mapping. Following values are valid :" \

    "<ul><li>true : linear mapping</li><li>false: uniform quantification</li></ul>

    HTML_HELP_CLOSE(),
  };
}
```

Then, we can add the parameters in the constructor by writing the following lines:

```
addParameter<DoubleProperty>("property",paramHelp[0],"viewMetric");

addParameter<int>("colormodel",paramHelp[1],"1");
addParameter<bool>("type",paramHelp[4],"true");
addParameter<Color>("color1",paramHelp[2],"(255,255,0,128)");
addParameter<Color>("color2",paramHelp[3],"(0,0,255,228)");
```

The picture below is the result of the sample of code above.


Tulip Parameter Editor

The following parameters are requested :

property: viewMetric

colormodel: 1

type: ✓

color1: 255 R 255 G 0 B 128 A

color2: 0 R 0 G 255 B 228 A

Move mouse cursor over a parameter label to display its documentation ...

6.1.2.2. Accessing a parameter

The class PropertyAlgorithm has a protected member called *dataSet* that contains all the parameters value.The DataSet class implements a container which allows insertion of values of different types and implements the following methods :

- `template<typename T> bool get (const std::string& name, T value) const` :Returns a copy of the value of the variable with name name. If the variable name doesn't exist return false else true.

- `template<typename T> bool getAndFree (const std::string& name, T value)` : Returns a copy of the value of the variable with name name. If the variable name doesn't exist return false else true. The data is removed after the call.

- `template<typename T> void set (const std::string& name, const T value)` :Set the value of the variable name.

- `bool exist (const std::string& name) const` :Returns true if name exists else false.

- `Iterator<std::pair<std::string, DataType> >* getValues () const` :Returns an iterator on all values

Has you could have guess, the one important to access a parameter is get() which allows to access to a specific parameter. Following is an example of its use :

```
DoubleProperty* metricS;
int colorModel;
Color color1;
Color color2;
bool mappingType = true;

if ( dataSet!=0 ) {
  dataSet->get("property", metricS);
  dataSet->get("colormodel", colorModel);
  dataSet->get("color1", color1);
  dataSet->get("color2", color2);
  dataSet->get("type", mappingType);
}
```

6.1.3. The PluginProgress class.

The class PluginProgress can be used to interact with the user. Following is a list of its members

6.1.3.1. Public members

Following is a list of all Public members :

- `ProgressState progress (int step, int max_step)` : This method can be used to know the global progress of or algorithm, the number of steps accomplished.
- `void showPreview (bool)` : Enables to specify if the preview check box has to be visible or not.
- `bool isPreviewMode ()` : Enables to know if the user has checked the preview box.
- `ProgressState state () const` : Indicates the state of the 'Cancel', 'Stop' buttons of the dialog
- `void setError (std::string error)` :Shows an error message to the user
- `void setComment (std::string msg)` :Shows a comment message to the user

6.1.3.2. PluginProgress example

In following small example, we will iterate over all nodes and notify the user of the progression.

```
unsigned int i=0;
unsigned int nbNodes = graph->numberOfNodes ();

const unsigned int STEP = 10;

node n;
forEach(n, graph->getInEdges(n))
{
  ...
  ... // Do what you want
  ...
  if(i%STEP==0)
  {
    pluginProgress->progress(i, nbNodes); //Says to the user that the algorithm h


    //exit if the user has pressed on Cancel or Stop
    if(pluginProgress->state() != TLP_CONTINUE)
    {
      returnForEach pluginProgress->state()!=TLP_CANCEL;
    }
  }
  i++;

}
```

Before exiting, we check if the user pressed stop or cancel. If he pressed "cancel", the graph will not be modified. If he pressed "stop", all values computed till now will be saved to the graph.

6.1.4. Example of a plugin skeleton

Following is an example of a dummy color algorithm (you can find more example in the tulip tarball):

```
#include <tulip/TulipPlugin.h>
#include <string>

using namespace std;
using namespace tlp;

/** Algorithm documentation */
// MyColorAlgorithm is just an example
/*@{*/


class MyColorAlgorithm:public ColorAlgorithm {
public:

  // The constructor below has to be defined,
  // it is the right place to declare the parameters
  // needed by the algorithm,
  // addParameter<DoubleProperty>("metric", paramHelp[0], 0, false);
  // and declare the algorithm dependencies too.
  // addDependency<Algorithm>("Quotient Clustering", "1.0");
  MyColorAlgorithm(PropertyContext& context):ColorAlgorithm(context) {

  }

  // Define the destructor only if needed
  // ~MyColorAlgorithm() {
  // }

  // Define the check method only if needed.
  // It can be used to check topological properties of the graph,
  // metric properties on graph elements or anything else you need.
  // bool check(string& errorMsg) {
  //   errorMsg="";
  //   return true;
  // }

  // The run method is the main method :
  //    - It will be called out if the pre-condition method (bool check (..)) returne

  //    - It is the starting point of your algorithm.
  // The returned value must be true if your algorithm succeeded.
  bool run() {
   return true;
  }
};
/*@}*/

// This line is very important because it's the only way to register your algorithm

// It automatically builds the plugin object that will embed the algorithm.

COLORPLUGIN(MyColorAlgorithm, "My Color Algorithm", "Authors", "07/07/07", "Commen

// If you want to present your algorithm in a dedicated submenu of the Tulip GUI,

// use the declaration below where the last parameter specified the name of submenu

// COLORPLUGINOFGROUP(MyColorAlgorithm, "My Color Algorithm", "Authors", "07/07/0
```

6.2. The Algorithm class.

The class Algorithm is the class from which your algorithm will inherits if you want to write a more general algorithm. Instead of modifying just a specific property, it can be used to modify the entire graph. In this section, we will list all its members to have a global overview on what we can use to develop such a plug-in.

6.2.1. Public members

Following is a list of all public members :

- `Algorithm (AlgorithmContext context)` :The constructor is the right place to declare the parameters needed by the algorithm. `addParameter<DoubleProperty>("metric", paramHelp[0], 0, false);` And to declare the algorithm dependencies.`addDependency<Algorithm>("Quotie Clustering", "1.0");`

- `~Algorithm ()` :Destructor of the class.

- `bool run ()` : This is the main method : - It will be called out if the pre-condition method (bool check (..)) returned true. - It is the starting point of your algorithm. The returned value must be true if your algorithm succeeded.

- `bool check (std::string)` : This method can be used to check what you need about topological properties of the graph, metric properties on graph elements or anything else.

The methods below, will be redefined in our plugin (See section plug-in skeleton).

6.2.2. Protected members

Following is a list of all protected members :

- `Graph* graph` :This graph is the one given in parameters, the one on which the algorithm will be applied.

- `PluginProgress* pluginProgress` : The class PluginProgress can be used to have an interaction between the user and our algorithm. See Section 6.1.3, "The PluginProgress class." for more details.

- `DataSet* dataSet` :This member contains all the parameters needed to run the algorithm. The class DataSet is a container which allows insertion of values of different types. The inserted data must have a copy-constructor well done. See the section called DataSet for more details.

6.3. Import plug-ins

In this section, we will learn how to create import plug-ins. Those plug-ins will inherit from ImportModule.

Following is a small description of the class ImportModule :

6.3.1. Public members

Following is a list of all public members :

- `ImportModule (AlgorithmContext context)` :The constructor is the right place to declare the parameters needed by the algorithm. `addParameter<DoubleProperty>("metric", paramHelp[0], 0, false);` And to declare the algorithm dependencies.`addDependency<Algorithm>("Quotie Clustering", "1.0");`

- `~ImportModule ()` :Destructor of the class.

- `bool import (const std::string name)` : This is the main method, the starting point of your algorithm. The returned value must be true if your algorithm succeeds.

The methods below, will be redefined in our plugin (See section plug-in skeleton).

6.3.2. Protected members

Following is a list of all protected members :

• `Graph* graph` :This graph is the one given in parameters, the one on which the algorithm will be applied.

• `PluginProgress* pluginProgress` : The class PluginProgress can be used to have an interaction between the user and our algorithm. See Section 6.1.3, “The PluginProgress class.” for more details.

• `DataSet* dataSet` :This member contains all the parameters needed to run the algorithm. The class DataSet is a container which allows insertion of values of different types. The inserted data must have a copy-constructor well done. See the section called DataSet for more details.

6.3.3. Skeleton an ImportModule derived class

```
 #include <tulip/TulipPlugin.h>
#include <string>

using namespace std;
using namespace tlp;

/** Import module documentation */
// MyImportModule is just an example
/*@{*/


class MyImportModule:public ImportModule {
public:

  // The constructor below has to be defined,
  // it is the right place to declare the parameters
  // needed by the algorithm,
  // addParameter<DoubleProperty>("metric", paramHelp[0], 0, false);
  // and declare the algorithm dependencies too.
  // addDependency<Algorithm>("Quotient Clustering", "1.0");
  MyImportModule(AlgorithmContext context):ImportModule(context) {
  }

  // Define the destructor only if needed
  // ~MyImportModule() {
  // }

  // The import method is the starting point of your import module.
  // The returned value must be true if it succeeded.
  bool import(const string &name) {
    return true;
  }
};
/*@}*/

// This line is very important because it's the only way to register your import mod
// It automatically builds the plugin object that will embed the algorithm.

IMPORTPLUGIN(MyImportModule, "My Import Module", "Authors", "07/07/07", "Comments

// If you want to present your algorithm in a dedicated submenu of the Tulip GUI,

// use the declaration below where the last parameter specified the name of submenu

// IMPORTPLUGINOFGROUP(MyImportModule, "My Import Module", "Authors", "07/07/07",
```

6.4. Export plug-ins

In this section, we will learn how to create export plug-ins. Those plug-ins will inherit from ExportModule.

Following is a small description of the class ExportModule :

6.4.1. Public members

Following is a list of all public members :

- `ExportModule (AlgorithmContext context)` :The constructor is the right place to declare the parameters needed by the algorithm. `addParameter<DoubleProperty>("metric", paramHelp[0], 0, false);` And to declare the algorithm dependencies.`addDependency<Algorithm>("Quotie Clustering", "1.0");`

- `~ExportModule ()` :Destructor of the class.

- `bool import (const std::string name)` : This is the main method, the starting point of your algorithm. The returned value must be true if your algorithm succeeded.

The methods below, will be redefined in our plugin (See section plug-in skeleton).

6.4.2. Protected members

Following is a list of all protected members :

- `Graph* graph` :This graph is the one given in parameters, the one on which the algorithm will be applied.

- `PluginProgress* pluginProgress` : The class PluginProgress can be used to have an interaction between the user and our algorithm. See Section 6.1.3, "The PluginProgress class." for more details.

- `DataSet* dataSet` :This member contains all the parameters needed to run the Algorithm. The class DataSet is a container which allows insertion of values of different types. The inserted data must have a copy-constructor well done. See the section called DataSet for more details.

6.4.3. Skeleton of an ExportModule derived class

```
#include <tulip/TulipPlugin.h>
#include <string>
#include <iostream>

using namespace std;
using namespace tlp;

/** Export module documentation */
// MyExportModule is just an example
/*@{*/


class MyExportModule:public ExportModule {
public:

  // The constructor below has to be defined,
  // it is the right place to declare the parameters
  // needed by the algorithm,
  // addParameter<DoubleProperty>("metric", paramHelp[0], 0, false);
  // and declare the algorithm dependencies too.
  // addDependency<Algorithm>("Quotient Clustering", "1.0");
  MyExportModule(AlgorithmContext context):ExportModule(context) {
  }

  // Define the destructor only if needed
  // ~MyExportModule() {
  // }

  // The exportGraph method is the starting point of your export module.

  // The returned value must be true if it succeeded.
  bool exportGraph(ostream &os,Graph *graph) {
    return true;
  }
};
/*@}*/

// This line is very important because it's the only way to register your export mod

// It automatically builds the plugin object that will embed the algorithm.

EXPORTPLUGIN(MyExportModule, "My Export Module", "Authors", "07/07/07", "Comments"

// If you want to present your algorithm in a dedicated submenu of the Tulip GUI,

// use the declaration below where the last parameter specified the name of submenu

// EXPORTPLUGINOFGROUP(MyExportModule, "My Export Module", "Authors", "07/07/07",
```

6.5. Compilation (Makefiles)

You can download plugins skeletons here[1] .

Following is a Makefile that compiles a plug-in :

```
 # Update the line below according to the tulip installation directory you choosed

TULIP_DIR=/usr/local/bin
TULIP_CONFIG=$(TULIP_DIR)/tulip-config
TULIP_VERSION=$(shell ${TULIP_CONFIG} --version)
LIB_EXTENSION=$(shell ${TULIP_CONFIG} --pluginextension)

# To limit, crash problems when loading a plugin library
# Tulip only load those whose names end with a compatible version number

TARGET= libmyalgorithm-$(TULIP_VERSION).$(LIB_EXTENSION)

# a plugin library may contain more than one algorithm
# so you can have several source files on the line below
SRCS = MyGeneralAlgorithm.cpp

CXX=g++
CXXFLAGS = -O3 -Wall -DNDEBUG `${TULIP_CONFIG} --cxxflags --plugincxxflags`

LDFLAGS=  `${TULIP_CONFIG} --pluginldflags` `${TULIP_CONFIG} --libs`

OBJS=$(SRCS:.cpp=.o)

DEPS=$(SRCS:.cpp=.d)

all: $(TARGET)

$(TARGET): $(OBJS)
 $(CXX) $(OBJS) -o $@ $(LDFLAGS) $(LIBS)

clean :
 -rm -f $(TARGET) *.o *.d

install: all
 install $(TARGET) `${TULIP_CONFIG} --pluginpath`

%.d: %.cpp
 $(CXX) -M $(CXXFLAGS) $< \
 | sed 's!\($*\)\.o[ :]*!\1.o $@ : !g' > $@; \
  [ -s $@ ] 2>/dev/null || rm $@

-include $(DEPS)
```

[1] http://tulip.labri.fr/samples/plugintemplates.tar.gz

Procedure 6.1. Compile and install your plugin

1.

```
[user@localhost ]$ make
```

Type make in a terminal.

2.

```
[user@localhost ]$ make install
```

Type make install in a terminal. Be careful that you may need root privileges to do this.

3.

```
[user@localhost ]$ make clean
```

’make clean’, will remove all generated files (libraries, *.o , ...)

Chapter 7. Tulip graph format

7.1. Nodes

The nodes are stored with a list of indices. The indices must be non negative integers.

Syntax

```
(nodes id_node1 id_node2 ...)
```

Sample :

```
(nodes 0 1 2 3 4 5 )
```

7.2. Edges

An edge is defined by providing three non negative integers. The first is the id of the edge, the second is the id of source node of the edge and the third is the id of target node of the edge.

Syntax

```
(edge id id_source id_target)
```

Sample

```
(edge 2 2 1)
```

It defines one edge with the node that has the id 2 as source and the node that has the id 1 as target.

7.3. Clusters

A cluster is defined by an integer which represent the cluster id, one string which is the name of the cluster(Two clusters can have the same name). Then it is define with a list of nodes and a list of edges. To define a subcluster we use the same method. One important point is that the id zero is reserved for the root graph (thus it cannot be used).

Syntax

```
(cluster id name
  (nodes id_node1 id_node2 ...)
  (edges id_edge1 id_edge2 ...)
  (cluster id name
    (nodes id_node1 id_node2 ...)
    (edges id_edge1 id_edge2 ...)
  )
)
```

Sample

```
(cluster 3 "cluster"
  (nodes 1 2 3 )
  (edges 2 8 )
  (cluster 4 "Sub Cluster"
    (nodes 1 2 )
    (edges 2 )
  )
)
```

7.4. Definitions of properties

The definition of properties is the following:

Syntax

```
(property cluster_id property_type "property_name"
  (default "default_node_value" "default_edge_value" )
  (node id value)
  ...
  (edge id value)
  ...
)
```

Sample

```
(property  0 bool "viewSelection"
  (default "false" "false" )
  (node 1 "true")
  (node 2 "true")
  (node 3 "true")
  (edge 2 "true")
  (edge 8 "true")
)
```

7.4.1. Property Type

- layout : This type enables to store nodes position in 3D. The position of nodes is defined by a set of 3 doubles `(x_coord,y_coord,z_coord)`. The position of edges is a list of 3D points. These points are the bends of edges. `((x_coord1,y_coord1,z_coord1)(x_coord2,y_coord2,z_coord2))`
- size : This type enables to store the size of elements. The size is defined with a sequence of three double. `(width,heigth,depth)`
- color : This type enables to store the color of elements. The color is defined with a sequence of four integer from 0 to 255. `(red,green,blue,alpha)`
- string : This enables to store text on elements.
- metric : This enables to store real on elements.
- bool : This type enables to store boolean on elements.
- int : This type enables to store integers on elements.

7.5. Properties of Tulip

`viewSelection`
type : bool, this property is the one used for selected elements in Tulip.

```
(property  0 bool "viewSelection"
  (default "false" "false" )
  (node 1 "true")
  (node 2 "true")
  (node 3 "true")
  (edge 2 "true")
  (edge 8 "true")
)
```

`viewLayout`
type : layout, this property is the one used for displaying graph in Tulip.

```
(property  0 layout "viewLayout"
  (default "(0,0,0)" "()" )
  (node 1 "(10,10,10)")
  (node 2 "(20,20,20)")
  (edge 1 "(15,15,15)(25,25,25)")
)
```

`viewColor`
type : color, this property is the one used for coloring graphs in Tulip.

```
(property  0 color "viewColor"
  (default "(235,0,23,255)" "(0,0,0,0)" )
  (node 1 "(200,0,200,255)")
  (node 2 "(100,100,0,255)")
  (node 3 "(100,100,0,255)")
  (edge 2 "(200,100,100)")
)
```

`viewLabel`
type : string, this property is the one used for labeling the graphs in Tulip(in label mode).

```
(property  0 string "viewLabel"
  (default "" "" )
  (node 1 "Hello")
  (node 2 "Bonjour")
  (node 3 "Bye")
  (edge 2 "Aurevoir")
)
```

`viewSize`
: type : size, this property is the one used for the size of elements displayed.

```
(property  0 size "viewSize"
  (default "(0,0,0)" "(1,1,1)" )
  (node 1 "(10,10,10)")
  (node 2 "(20,20,20)")
)
```

`viewShape`
: type : int, this property is used for defining the shape of elements.

```
(property  0 int "viewShape"
  (default "0" "0" )
  (node 1 "1")
  (node 2 "2")
)
```

`viewTexture`

`viewMetaGraph`

`viewRotation`

7.6. GlScene

Scene structure are stored with a xml structure.

First, scene's data (viewport and background color) are store in data element

Syntax

```
<data>
  <viewport>(x,y,width,height)</viewport>
  <background>(red,green,blue,alpha)</background>
</data>
```

After, scene children are store in children element

GlLayer and GlComposite have two elements : data and children

All others glEntities have one element : data

The scheme of glScene xml are (for example) :

```
<scene>
  <data>
    ...
  </data>
  <children>
```

```
    <LayerName1 type="GlLayer">
      <data>
  ...
      </data>
      <children>
  ...
      </children>
    </LayerName>
    <LayerName2 type="GlLayer">
      <data>
  ...
      </data>
      <children>
  <EntityName type="EntityType">
    <data>
      ...
    </data>
  </EntityName3>
      </children>
    </LayerName2>
   </children>
 </scene>
```

For glEntities data see .cpp file

Chapter 8. Programming Guidelines

8.1. Generalities

The presentation of a program indicates the quality of programming. This section relates to the common recommendations for the Tulip project. Each new programmer has to follow the expressed rules.

In the header files, the programmer should write a headline containing : his name with personal email adress (for students), the date of the last modifications, a reminder of the licence GPL[1] and the references of the code (for example, an algorithm). Header files must include a construction that prevents multiple inclusions. The convention is an all uppercase construction of the file name, the h suffix and prefixed by "Tulip", separated by an underscore.

```
#ifndef Tulip_MYTYPE_H
#define Tulip_MYTYPE_H

...

#endif // Tulip_MYTYPE_H
```

The organisation of files must be comprehensible. New module leads to a new set of files : a `*.cpp` and a `*.h` named with the name of the type. If the structure implicates that all methods are inline, the creation of a `.cxx` file is better than a `.cpp` file. The `cxx` should be included at the bottom of the header file. None implementation is in the header file. In the Tulip hiearchy, the cxx files are in a directory "cxx" in the header location.

The indentation is an important part for a easy reading in a file and a best understanding. Code must be properly indented to show the syntactic structure of the program. It is useless to space out excessively the code. A conventional indentation is just necessary. None useless **TAB** or spaces. The { caracter for the opening of a method or a function must be at the end of the line, not in following line. Each new fitted block of program implies a new shift for the indentation.

Each new module inserted in the Tulip library must be included in the namespace `tlp`. It is necessary in order to prevent eventually incompatibilities.

```
namespace tlp{

/* code inserted */

}
```

Tulip is dependent of the STL[2]. It provides a set of performing objects that you should use : vector, map, string, ... It exists two ways to use it. In the `.h` or `.cxx` files, you should preface them with the `std` namespace (e.g. `std::string s;`). You will refer them with the fullname : namespace and class name. For the `.cpp` files, you can use the short name if you insert the line at the top of your document `using namespace std;`.

In a header file (`.h` or `.cxx`) :

[1]General Public License
[2]Standard Template Library

```
class MyClasse{
 public:
   Myclasse();
   ~MyClasse(){}
   void draw();

 private:
   std::string mystring;
};
```

In a source file (`.cpp`) :

```
using namespace std;

MyClasse::MyClasse(){
   mystring = "Hello world";
}
void MyClasse::draw(){
   cout<<mystring<<endl;
}
```

8.2. Naming conventions

Programmer of Tulip has to follow some rules for choosing Type, Functions, or Variables names. Each names must be in English and choose to an easy understanding, descriptive and accurate. Each important word must be found in the name.

List of Rules

- Types (struct, class, ...) : Names must be in mixed case starting with upper case. Each word should have first letter in upper case. Don't use underscore to separate words. (e.g. `SortedVector, OrientedList, RedBird, ...`)
- Functions and Methods : Names must be in mixed case like for Types, but starting with lowercase. After the first one, each important word has first letter in upper case. Don't use underscore to separate words. (e.g `drawString(), computeFormulas(), ...`)
- Variables : Names must be in lower case. (e.g. `instances, nodes, tableofcontents, ...`)
- Constants : Names must be in upper case. (e.g. `MAXSIZE, BLACKCOLOR, ...`)
- Macro and Enumeration constants : Names must be defined in upper case with an underscore between words. (e.g. `LAYOUTPLUGIN(C, N, A, D, I, V, R), ...`)
- Namespaces : Names must be in lower case. (e.g. `tlp`)

The *setter* and *getter* must begin with the keyword `set` or `get`. All of the methods or functions should begin with a verb for understanding its goal. The prefix of a boolean variables or methods should be *is, can, has, should* : `bool isValid(const edge e) const`, function specified if the edge is valid.

8.3. Code Comments

All of the comments in the source and the header files must be written in the English language. Adding a lot of comments is a simple way to leave a clear code for the next programmer who will receive your work. A part of comments must be written with several rules to help the documentation of your work generating automatically with *Doxygen*. Choose a part of your comments to describe your work. See Section 9.4, "Code documentation". It is important that a programmer can understand the using of your work and how to use your work.

Before a declaration of a class, you should write a little description to explain it. The role, the pre-requisites, the post-requisites, the return values and the parameters should be written before the declaration of the function or method. It is the minimum for an easy comprehension of your work. In the code, it is useless to comment each line because the comments are often a paraphrase of the code. It just is essential to write a comment for strong parts that you have thought.

8.4. Integration in Tulip project

The build of Tulip uses a mechanism of the GNU operating system. GNU has several tools used for the management of the configuration files. It modifies the `makefile` to adapt them to the distribution you have and the tools you need : the most important tools are autoconf, automake and libtool. Tulip generates three libraries : `libtulip, libtulip-ogl, libtulip-qt`, a software : `tulip` and a script : `tulip-config`. In a thirdparty, Tulip compiles several external libraries needed by the software : ftgl, gle, gzstream, triangle.

8.4.1. GNU Build system

8.4.1.1. Presentation

The goals of this system is to simplify the development of portable programs and the building of programs that are distributed as source code.

Autoconf is a tool of GNU producing shell scripts that automtically configure software packages to adapt to many kinds of UNIX systems. It is not the unique tool, it runs with others to solve all problems to making portable software. It generates configurations files : specially the `configure` script from a `configure.in` or `configure.ac` file. Running this script, you produce the customized Makefiles, and other files. It checks for the presence of each feature that the software need. Autoconf requires GNU M4 in order to generate the scripts.

To this end, GNU has developed a set of integrated utilities to finish the job of Autoconf. Automake is the next in run. It is a tool for generating `Makefile.in` from files called `Makefile.am`. Each `Makefile.am` is basically a series of `make` variable definitions, with the GNU Makefile standards. Automake requires Autoconf in order to be used properly.

The last is Libtool. It makes it possible to compile position independent code and build shared libraries in a portable manner. It does not require either Autoconf, or Automake and can be used independently. Automake however supports libtool and operates with it in a seamless manner.

8.4.1.2. A simple example

... to understand the basic mechanism.

To create a `configure` script with autoconf, you need so to write an autoconf input file `configure.ac` (or `configure.in`, use in previous versions of Autoconf). In this example, it is created a `configure.ac` file but Tulip contains `configure.in`. The both files are correct.

```
`hello.c`
 #include <stdio.h>
 main(){
   printf("Hello world!\n");
 }

`Makefile.am`
 bin_PROGRAMS = hello       ❶
 hello_SOURCES = hello.c     ❷

`configure.ac`
 AC_INIT(hello, 1.0)        ❸
 AC_CONFIG_SRCDIR(hello.c)  ❹
 AM_INIT_AUTOMAKE()         ❺
 AC_PROG_CC               ❻
 AC_OUTPUT(Makefile)        ❼
```

❶ `bin_PROGRAMS` : specifies the name of programs that are building.
❷ `hello_SOURCES` : specifies the sources code that composed the program “hello”.
❸ `AC_INIT` : initializes the `configure` script. It must be passed as argument the name of the package and the version.
❹ `AC_CONFIG_SRCDIR` : specifies a file in the source directory. `configure` script checks for the existence of this file to make sur that directory that it is told contains the source code in fact does. Any source file could do.
❺ `AC_INIT_AUTOMAKE` : performs some further initializations that are related to the fact that we are using Automake. If you are writing your `Makefile.in` by hand, then you do not need to call this command.
❻ `AC_PROG_CC` : checks to see which C compiler you have
❼ `AC_OUTPUT` : tells the configure script to generate `Makefile` from `Makefile.in`

Create the files and Run :

```
$ aclocal
$ autoconf
```

The `aclocal` command installs a file called ‘aclocal.m4’. It contains the knowned Autoconf macros to be in use in `configure.ac`, like `AC_PROG_CC`. If you want to include your macros, you can create an `acinclude.m4` file. An other cache directory is created to store the traces of the runs of `m4`. It is called `autom4te.cache`.

The `autoconf` command create the `configure` script. Then, Run :

```
$ automake -a
```

It displays :

```
configure.ac: installing `./install-sh'
configure.ac: installing `./missing'
```

```
Makefile.am: installing `./INSTALL'
Makefile.am: required file `./NEWS' not found
Makefile.am: required file `./README' not found
Makefile.am: required file `./AUTHORS' not found
Makefile.am: required file `./ChangeLog' not found
Makefile.am: installing `./COPYING'
Makefile.am: installing `./depcomp'
```

This creates copies of `install-sh`, `missing`, `COPYING`, `depcomp`. These files are required to be present by the GNU coding standards. But `NEWS`, `README`, `AUTHORS`, `ChangeLog` are not generated. You have to create them. If you have not them and you attempt to do `make distcheck`, then it will deliberately fail. To create it :

```
$ touch NEWS README AUTHORS ChangeLog
```

Then, you have to run `automake -a` a second time. This one has created a `Makefile.in` file from `Makefile.am`. In this file, we have specify what are building and the used sources. For a library, you should define the `lib_LIBRARIES` variable.

Now the package is exactly in the state that the end-user will find it when person unpacks it from a source code distribution. To test you program, you can write :

```
$ ./configure
$ make
$ ./hello

and ...

$ make install
$ make uninstall
$ make dist
...
```

8.4.2. File adds

To integrate a new module, set of types, in the Tulip project, you must to know which library is concerned : General library, OpenGL library, QT library, ... For each case, the procedure is the same. `tulip/library/tulip-ogl/` is the directory to integrate a library attached to the Opengl library. All of the `.cpp` files are pasted in the `src` subdirectory, the `.h` files in `include/tulip` and `cxx` files in `include/tulip/cxx`. Some modifications of your code should be necessary. The inclusion of files of Tulip project (included your work) is made with < and > because the compiler knows the path. For Tulip, the header files is in a special directory : `tulip`.

`#include <tulip/TheFile.h>` is an exemple of the inclusion.

So, you have to modify two files in the directory of your library to indicate the new files. `include/Makefile.am` is the first. You have to complete a variable containing all `.h` and `.cxx` files with your header files named `nobase_include_HEADERS`. This name is a choice for the processing of the GNU build system. The second one is `src/Makefile.am` and so, you complete the variable containing all `.cpp` files with your source files : `libtulip_ogl_la_SOURCES`, `libtulip_la_SOURCES` or `libtulip_qt_la_SOURCES` depending of the librairie you complete. You have modified the both `Makefile.am` but the Makefile not. To update it, you have to recreate the `configure` file at the root directory and run it again. To do it, run `./gen-conf` and

./configure. To avoid this procedure at each modification of the Makefile.am, you can specify an option when you use configure script : --enable-maintainer-mode. See Section 2.1, “Options”, for more details about the options. Now, the next compilation includes your work.

8.4.3. Compilation directives : Makefile.am

If you want to change the directive of compilation for a program or a library, then you have to complete or modify the variables attached to the program or library. This section gives the essential variables (with their forms) for a customized compilation.

Makefile.am can use the same syntax as with ordinary makefiles. General variable can be defined, available for all your building objects.

`INCLUDES = -I/dir1 -I/dir2 -I$(top_srcdir)/src...`
: Insert the -I flags that you want to pass to your compiler when it builds executables.

`LDFLAGS = -L/dir1 -L/dir2 ...`
: Lists all the library files that will be compiled with make and installed with make install under prefix/lib.

`LDADD = MyClasse.o ...  $(top_builddir)/dir1/libmylib.la ... -lmylib ...`
: List a set of object files, uninstalled libraries and installed libraries that you want to link in with all of your executables. Please refer to uninstalled libraries with absolute pathnames. Because uninstalled libraries are built files, you should start your path with $(top_builddir). There is a set of variables like top_builddir which are defined by configure when it processes a Makefile and they can be used in all others variables presented here.

`srcdir`
: The relative path to the directory that contains the source code for that Makefile.

`abs_srcdir`
: Absolute path of srcdir.

`top_srcdir`
: The relative path to the top-level of the current build tree. In the top-level directory, this is the same as srcdir.

`abs_top_srcdir`
: Absolute path of top_srcdir.

`builddir`
: Rigorously equal to “./”. Added for the symmetry only.

`abs_builddir`
: Absolute path of builddir.

`top_builddir`
: The relative path to the top-level of the current build tree. In the top-level directory, this is the same as builddir.

`abs_top_builddir`
: Absolute path of top_builddir.

The following targets are used for a special directory or for special buildings:

`bin_PROGRAMS = prog1 prog2 ...`
: Lists the executable files that will be compiled with `make` and installed with `make install` under `prefix/bin`, where `prefix` is usually `/usr/local` but you can specify to an other value.

`lib_LIBRARIES = lib1.la lib2.la ...`
: Lists all the library files that will be compiled with make and installed with `make install` under `prefix/lib`.

`SUBDIRS = dir1 dir2 ...`
: Lists all the subdirectories that we want to build before building this directory. `make` will recursively invoke itself in each subdirectory before doing anything on the current directory. If you mention the current directory "." in `SUBDIRS` then the current directory will be built first, and the subdirectories will be build afterwards.

`EXTRA_DIST = file1 file2 ...`
: Lists any files that you want to include into your source code distribution.

`include_HEADERS = fich1.h fich2.h ...`
: Lists all the public header files in this directory that you want to install to `prefix/include`. If you change the keyword `include` by `noinst`, then you can specify headers that will not be installed.

For each progam, a set of variables should be declared :

`prog_SOURCES = fich1.c fich2.c ...`
: Lists all the files that compose the source code of the program. Header files can be specified here.

`prog_LDADD = $(top_builddir)/dir1/lib1.a -lext1 -lext2 ...`
: Lists the libraries that need to be linked with your source code. Installed libraries should be mentioned using '-l' flags. Uninstalled libraries must be mentioned using absolute pathnames. Please use `$(top_buiddir)` to build a path to a directory.

`prog_LDFLAGS = -L/dir1 -L/dir2 -L/dir3 ...`
: Adds the '-L' flags that are needed to find the installed libraries that you want to link in `prog_LDADD`.

`prog_DEPENDENCIES = dep1 dep2 dep3 ...`
: Lists any targets that you want to build before building this program.

In the same way, you can specify variable for a specialy library or shared library by prefixing the variable by the name of the library.

Occasionally it is useful to know which Makefile variables that Automake uses for compilations. For instance you might need to do your own compilation in some special cases. Some variables are inherited from Autoconf; these are CC, CFLAGS, CPPFLAGS, DEFS, LDFLAGS, LIBS, CXX, CXXFLAGS, ... There are some additional variables which Automake defines itself:

`AM_CPPFLAGS`
: The contents of this variable are passed to every compilation which invokes the C preprocessor; it is a list of arguments to the preprocessor. For instance, `-I` and `-D` options should be listed here. Automake already provides some `-I` options automatically. `AM_CPPFLAGS` is ignored in preference to a per-executable (or per-library) `_CPPFLAGS` variable if it is defined.

`INCLUDES`
: This does the same job as `AM_CPPFLAGS`. It is an older name for the same functionality. This variable is deprecated; we suggest using `AM_CPPFLAGS` instead.

`AM_CFLAGS`
: This is the variable which the Makefile.am author can use to pass in additional C compiler flags. It is more fully documented elsewhere. In some situations, this is not used, in preference to the per-executable (or per-library) `_CFLAGS`.

`COMPILE`
: This is the command used to actually compile a C source file. The filename is appended to form the complete command line.

`AM_LDFLAGS`
: This is the variable which the Makefile.am author can use to pass in additional linker flags. In some situations, this is not used, in preference to the per-executable (or per-library) `_LDFLAGS`.

`LINK`
: This is the command used to actually link a C program. It already includes -o $@ and the usual variable references (for instance, `CFLAGS`); it takes as “arguments” the names of the object files and libraries to link in.

8.4.4. Variable prefix

`nobase_`
: e.g. `nobase_include_HEADERS`, mentionned that all the headers files will not be installed in the same place. It is possible to make subdirectories. `nobase_` should be specified first when used in conjunction with either `dist_` or `nodist_`.

`noinst_`
: denotes data which do not need to be installed.

`dist_ /nodist_`
: denotes files or data that will be included to the distribution (or not with `nodist_`).

8.4.5. References

Developing software with GNU : the GNU build system - http://www.amath.washington.edu/~lf/tutorials/autoconf/toolsr

Autoconf Manual - http://www.gnu.org/software/autoconf/manual/

Automake Manual - http://www.gnu.org/software/automake/manual/

Chapter 9. Documentation Guidelines

9.1. Definitions

1. Docbook is a collection of standards and tools for technical publishing. A Docbook file is composed of SGML(3) tags and is also dependant of a Document Type Definition(2). Docbook has defined standard DTD that you can find in the docbook tools.

2. DTD : Document Type Definition. The DTD defines the vocabulary of content elements that an author can use and how they relate to each other. For example, a book element can contain a title element, any number of para elements for paragraphs, and any number of chapter elements.

3. SGML : Standard Generalized Markup Language, XML : Extensible Markup Language

4. XSL (Extensible Stylesheet Language) is a language of description used for the transformation of the sgml file into formatted ouput: XSL processors like xsltproc, saxon or xalan, ... do that.

5. Catalog : In XML, it provides a mapping from generic addresses to specific local directories on a given machine. A catalog can be used to locate the DocBook DTD, system entity files, and stylesheet files during processing.

9.2. Tools installation

9.2.1. Docbook XSL Stylesheet

There is a great reference for installing tools : Docbook XSL : the guide[1].

To write a manual using Docbook and the XSL stylesheets, we need four tools :

- Docbook DTD
- Docbook XSL stylesheets
- XSL processor
- XSL-FO processor (optional)

[1] http://www.sagehill.net/docbookxsl/ToolsSetup.html

There are RPM (docbook-dtds, docbook-style-xsl), Debian (docbook-xml, docbook-xsl) packages for Linux systems, Fink packages for Mac systems, and Cygwin and other packages for Windows systems. The installation of Docbook and the XSL stylesheet create a file named `/etc/xml/catalog`. It contains the informations about all others catalogs to resolve the path of the DTDs and the stylsheets from the URI.

9.2.1.1. Manual Installation : Docbook XSL stylesheet

As we have some problems with the DTD provided by the distributions, Tulip gives the DTD 4.4 of Docbook XML and is located in `$TULIPDIR/docs/common/dtd`. So, this section is for informations. The XSL packages seem to have great catalog resolutions, but if you have problems, this section can help you.

If you need another solution or you have problems for the compilation, you can download the source package on oasis web site[2] for Docbook and on the SourceForge page[3] of XSL stylesheets. The Docbook sources are in a `zip` file. Unzip your package in a directory in the place you prefer. The Docbook XML DTD consists of a main file `docbookx.dtd`, several module files and a catalog file, named `catalog.xml`. To update the changes of location of your DTD, edit the file `CatalogManager.properties` in `$TULIPDIR/docs/common/`. Add the catalog file with its asbolute path to the `catalogs` variable.

```
CatalogManager.properties :

catalogs=/etc/xml/catalog;/home/bardet/docbook-4.4/catalog.xml
relative-catalogs=false
static-catalog=yes
catalog-class-name=org.apache.xml.resolver.Resolver
verbosity=1
```

Concerning the XSL stylesheet package, you just have to do the same thing. You can execute the script of intallation. You have to add the reference of the catalog too. Edit the file `catalog.xml` and complete the `catalogs` variable with the reference of the catalog of your XSL stylesheet. This catalog resolution make sure that the path, the differences could be find in all Operation Systems.

9.2.1.2. XSL processor

Currently, there are three to do XSL Transform processing with the recommendations of XSL : saxon (http://saxon.sourceforge.net/), xalan (http://xml.apache.org/xalan-j/), and xsltproc (http://xmlsoft.org/XSLT/xsltproc2.html). For the Tulip project, we have decides to use Saxon for the XSL processor. Xsltproc is the xslt c library for gnome. This program is a way to use the library *libxslt* with a command line tool for applying XSLT stylesheets to XML documents. This application runs quickly but does not include the implementation of the extensions. Xalan is an other solution but Saxon is the most recommanded.

Saxon is a free processor written in Java, so it can be run on any operating system with a modern Java interpreter. To install it, you have to download on sourceforge the last release 6.5.4. for debian, it exists a deb package. This is the full version that implements the XSLT 1.0 standard. It runs on Java system and provides opportunities for extensions. For Windows, it exists the Instant Saxon a precompiled version that runs only on Microsoft Windows. Saxon is distributed as a zip package. You have to unzip it into a suitable location. It gives three `.jar` files ; `saxon.jar` contains the XSLT processor. In according to the location, you have to update your CLASSPATH. You need to include the full path to the necessary .jar files in the CLASSPATH environment variable. To update it :

```
CLASSPATH=$CLASSPATH:/usr/saxon/saxon.jar:\
```

[2] http://www.oasis-open.org/docbook/xml/
[3] http://sourceforge.net/projects/docbook/

```
/usr/docbook-xsl/extensions/saxon653.jar
export CLASSPATH
```

If your CLASSPATH is incorrect, you get an error message about `NoClassDefFoundError`. To generalize it, you have to change your `.bashrc` or `.bash_profile`. In this example, a second package is included : `/usr/docbook-xsl/extensions/saxon653.jar`, in Fedora Core 4, it is located in `/usr/share/sgml/docbook/xsl-stylesheets/extensions/`. It contains the implementation of the extensions of XSL Transformations. This extensions are necessary to manage some tags and are included in the docbook tools. The general syntax for compilation is like followed :
`java com.icl.saxon.StyleSheet [options]` [file.docbook] [file.xsl] `[param=value]`
However, Tulip provides the sources of saxon and you do not worry to this installation to avoid the complicated installation.

Warning

This system of catalog resolution is made for finding located files. With the packages of Fedora and perhaps with others distribution, the catalogs include a declaration with URL adresses. So the catalog resolving processor try to connect on the Net.

```
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD XML Catalogs V1.0//EN"
"http://www.oasis-open.org/commitees/entity/release/1.0/catalog.dtd">
```

It should be erased in all catalog files which blocks the offline compilation (/etc/xml/catalog, /usr/shar/sgml/docbook/xmlcatalog, ...).

9.2.1.3. XSL-FO processor for PDF output

To transform the docbook sources in pdf documents, we need to use an other format FO. It is running with a stylesheet in docbook-xsl package. The role of the XSF-Fo processor is to transform FO files in printable files, like pdf. You have to install passivetex. Fedora and Debian includes this package, but you can download and install it with its web site, http://www.tei-c.org.uk/Software/passivetex/. It exists a program named `pdfxmltex` giving tex macros for the XSL-Fo processing.

9.2.2. Doxygen

It can already be in your distribution. For distributions like Debian or Fedora Core, you can used :
`yum [install` doxygen]
`apt-get [install` doxygen]
For another solution, you can find source files and CVS repository in the web site of Doxygen (http://www.stack.nl/~dimitri/doxygen).

9.3. Handwriting for the manuals

Some works on the Tulip project lead to make a part of a handbook for developer or users. Docbook utilities are used to create them.

Docbook is a DTD (Document Type Definition) for XML or SGML document that define elements. It is a kind of grammar of the source document. This is in relation with stylesheets (DSSSL [1] or XSL) which allow you to publish on the Web (in HTML, XHTML) on a printable documentation (PS, PDF, RTF...) with your Docbook documents. The main advantages of the docbook format is the possibility to seperate the contents of the forms. The tools which can transform the source documents are generic and can pratically run on all Operation Systems. Docbook is the successor of LinuxDoc.

For Tulip documentation, the source documents are in XML, because XML leads to take the place of SGML to alleviate compatibility problems with browser software. It's a new, easier version of the standard rules that govern the markup itself. To find all of the xml elements to composed your own document, you can see : http://www.docbook.org/tdg5/en/html/pt02.html

For each document, a set of XML files should be created. To help the editing of the handbook, it is possible to use general editors like kate, emacs, quanta. See Docbook Editors[4]. To configure Kate, select `Configure Kate` -> `Settings` -> `Configure Kate`. Select the plugin item from the application tree and check the `Kate XML Completion`, and the `Kate XML Validation` boxes.

To compile the handbooks, you just have to type `make html`. If you want to test the validation of the sources of the handbooks, type `make check`. The last command is `make pdf`, it produces the printable output.

1. DSSSL: Document Style Semantics and Specification Language. For more informations, you can visit the web site: http://www.jclark.com/dsssl/

9.3.1. Some Tricks

```
<?xml version='1.0'?>                                 ❶
<!-- My first Docbook file -->                        ❷
<!DOCTYPE book PUBLIC "-//OASIS//DTD Docbook XML V4.4//EN" ❸
   "http://www.oasis-open.org/docbook/xml/4.4/docbookx.dtd">
```

❶ defined that it is a XML document
❷ the comments are writing between <!-- and -->
❸ DTD, Document Type Declaration with pulic identifier

The identification of the DTD is used by the document to know which root element is. The declaration could have a different aspect if you use the system identifier : `<!DOCTYPE book SYSTEM "http://www.oasis-open.org/docbook/xml/4.4/docbookx.dtd">` Here, “OASIS” is the owner, the declaration is “DTD Docbook XML V4.4” and the language is English “EN”. The keyword “book” specifies that the root element is *book*. An example of the structure of a Docbook file is like followed.

```
<!DOCTYPE book SYSTEM
   "http://www.oasis-open.org/docbook/xml/4.4/docbookx.dtd">
<book>
   <bookinfo>
   <title>My First Book</title>

      <author><firstname>Jean-Pierre</firstname><surname>DUPONT
               </surname></author>
      <copyright><year>2005</year>
                 <holder>Jean-Pierre DUPONT</holder>
        </copyright>
   </bookinfo>
   <preface> ... </preface>
   <chapter> ... </chapter>
      <sect1> <title>..</title>
         <para> ...</para>
```

4 http://i18n.kde.org/doc/doc-primer/docbook-editors.html

```
      </sect1>
   <chapter> ... </chapter>
   <chapter> ... </chapter>
   <appendix> ... </appendix>
   <appendix> ... </appendix>
   <index> ... </index>
</book>
```

Each blocks of text is in a *para* element.

9.3.1.1. Validation

The validity of a document is very difficult to detect, so we use a program to do that. A validating parser is a program that can read the DTD and determine whether the exact order of elements in the document is valid according to the DTD.

To determinate if the XML code is valid, the program xmllint can be used with the option --valid. This program is a parser dependent of the library *libxml2*.
`xmllint [--valid  --noout` myfile.docbook]
When there are errors in the document, xmllint detects the number line and displays the form that it should be.

```
index.docbook:485: element sect2: validity error : Element sect2
content does not follow the DTD, expecting (sect2info? , (title ,
subtitle? , titleabbrev?) , (toc | lot | index | glossary |
bibliography)* , (((calloutlist | glosslist | bibliolist | itemizedlist

| orderedlist | segmentedlist | simplelist | variablelist | caution |
important | note | tip | warning | literallayout | programlisting |
programlistingco | screen | screenco | screenshot | synopsis |
cmdsynopsis | funcsynopsis | classsynopsis | fieldsynopsis |
constructorsynopsis | destructorsynopsis | methodsynopsis | formalpara

| para | simpara | address | blockquote | graphic | graphicco |
mediaobject | mediaobjectco | informalequation | informalexample |
informalfigure | informaltable | equation | example | figure | table |
msgset | procedure | sidebar | qandaset | task | anchor | bridgehead |
 remark | highlights | abstract | authorblurb | epigraph |indexterm |
beginpage)+ , (refentry* | sect3* | simplesect*)) | refentry+ | sect3+ |

 simplesect+) , (toc | lot | index | glossary | bibliography)*), got (
refentry)
</sect2>
          ^
make: *** [check] Erreur 1
```

It exists several attempts to produce an print or web publishing. Recently, the XML workgroup has made a standard Extensible Style Language (XSL). It provides a set of stylesheet able to transform the Docbook documents. DSSSL is an other possibility, but we use the XSL stylesheet for the Tulip project. This stylesheet is an argument for the compilation with an XSL processor for the XSL transformations.

9.3.2. Docbook FAQ

9.3.2.1. how to separate the work in several files ?

The first solution to write a part of a handbook is to complete the main document. To avoid that several persons work on the same file, it exists a solution to cut the docbook file. You can write a section or a chapter and you just have to include your part in the main file. To create a valid file for the compiler, you shoud modify the head of a docbook file and specify the kind of document you do. For a chapter, your file is like followed :

```
<!DOCTYPE chapter PUBLIC "-//OASIS//DTD Docbook XML V4.4//EN">
<chapter>
   <title>My personal Chapter</title>
   <sect1> ... </sect1>
</chapter>
```

The identification gives the kind of file you write. For including your work, you just have to complete the declaration of the main file and insert a special tag located at the place that you want the inclusion. Don't forget to erase the declaration of your work. The file must begin with the markup "chapter".

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD Docbook V4.4//EN" [
  <!ENTITY myPersonalFile SYSTEM "myFile.xml">
]>
<book>
...
</chapter>
&myPersonalFile;
<chapter>
...
```

9.3.2.2. How to insert a figure ?

the element uses for the insert of a figure is "<figure>". It enables the numerotation of the figures. Warning, the width of the figures should not exceed a certain value to avoid that it is truncated in the pdf transform. You can specify a title and other parameter in the limits of the DTD.

```
<figure> <title>Screenshot of the result</title>
<graphic fileref="doxygen-ex.png"/></figure>
```

In order to insert a figure as "inline", use the tag `inlinegraphics`

9.3.2.3. How to make annotation in a program listing (or in a console listing) ?

See DocBook XSL : the complete guide[5]

To do that, you use the set of tags `programlisting, co, calloutlist, callout.`

```
<programlisting>
class MyClasse{ <co id="myid" linkends="mylink"/>
};
</programlisting>
<calloutlist>
   <callout arearefs="myid" id="mylink">
      <para>... comments ...</para>
   </callout>
</calloutlist>
```

[5] http://www.sagehill.net/docbookxsl/AnnotateListing.html#Callouts

The `programlisting` contains the program that you want to display. It is a verbatim mode. Warning, special caracters could be not allow : <, for example. The `code` element represents the callout you want to place. The location of the insertion is important because it is the location on display. Two attributes are essential : `id` for the identification and `linkends` for indicate which comments is linked with the callout. After the `programlisting` element, we use a `calloutlist` element to define a list of comments. each comment is contained in a `callout` element. It has attributes whose `arearefs` to make a reference to callouts and id for the identification.

It exists an other solution but it is more difficult. You have to specify coordinates of place where you want a callout. This solution is an easy and quick way to comment your program listing.

```
class MyClasse{ ❶
};
```

❶ ... comments ...

To adapt the case to the console listing, you should use the `screenco, screen, callout, ...` elements.

9.4. Code documentation

9.4.1. Presentation

Doxygen is a documentation system for several languages like C++. It can generate an on-line documentation browser (in HTML), a set of manpages, an off-line reference manual (in LaTeX) and/or others from the set of documented source files of your project. This kind of documentation is a necessary tool for developers to find informations about the code. It is not a way to explain how it is imagined but which possibilities you have. It is available for several distributions like Fedora Core, Debian, Gentoo and others,.. Windows, MacOs and Sun Solaris too.

To create your first Doxygen documentation, you need a config file .cfg or .doxygen. To generate it :
`doxygen [-g` myfile.doxygen]
The file contains the options you can choose for the documentation generation. Comments indicates how to use this variables. The INPUT variable contains the files or directories to find the source files. When you have set all tags with good values, you can generate the documentation. Doxygen checks the source files to extract the informations in special comments and tags. See this page for the informations about the variables : http://www.stack.nl/~dimitri/doxygen/config.html. To create it :
`doxygen` [myfile.doxygen]
By default, it creates directories : html, latex, and/or man, ...

For the Tulip project, It exists two documentations. One is destinated for the Developer team and this other one is for the user of the librairies. The Configuration files are generated by the Makefile processing.

9.4.2. Developer comments

The code documentation, generated by Doxygen is completely dependant of the developer comments. It is important that developers follow the grammar rules.

So the blocks of documentation in the sources files are the C++ comment blocks. For each item of code, there are two types of descriptions, which together form the documentation : a brief description and detailed description. A detailed description is used to explain generously the existence of an item.

```
/**
* detailed description
```

```
*/

or

/*!
* detailed description
*/

or

/*!
 detailed description
*/

or others ....
```

To make a brief description, you can use the command `\brief`. This command ends at the end of a paragraph, so the detailed description follows after an empty line. An other option is to use a special C++ style comment which does not span more than one line.

```
/*! \brief Brief description.............
*       ...............
*
*  Detailed description
*/

or

/// Brief description
/** Detailed description. */

or

//! Brief descripion.

//! Detailed description
//! starts here.

or others ....
```

For more details, the web site of Doxygen[6] explains it. In general, the comments must be before the declaration or definition of a type, a function or a member. If you want to putting the documention after, you have to add a marker <. Note that you can place your comment at other places using some tags like `\class, \union, \fn, \var,` ... or `@class, @union, @fn, @var,` ...

It exists several tags to help you for commenting and writing a description : all of it begin with a backslash \ or an at-sign `@`.

- `@author`, name of the author.
- `@param`, to write a special comment on a parameter of a method or function
- `@see`, to make a reference to an other object or function

[6] http://www.doxygen.org/docblocks.html

- `@return`, to indicate the exit of a function
- `@date`, date of creation
- `@note`, describe a role
- `@attention`, write a caution
- `@warning`, write a warning
- `@pre`, write a prerequisite
- `@remark`

The complete list is on this page[7] in the Doxygen web site.

Example 9.1. Doxygen : A simple source file

[7] http://www.stack.nl/~dimitri/doxygen/commands.html

```
// ... comments not include  ...
///  A example of class : MyClass.
/**
   a more detail class description :
   \author Me
   \date 29/07/2005
*/
#include <string>
class MyClass
{


   /* ... comments not include ... */
   public:
      /** the constructor of the class */

      /**  the detail description of the constructor. */
      MyClass(){i=0;}

      //! A destructor.
      ~MyClass(){}

      /// drawing of a string
      /**
         \param s the string to display
         \return there is no return
         @sa MyClass(), ~MyClass()
      */
      void draw(const char *s="Hello World");

      /* exemple of doc comments not before the declaration */
      unsigned int getI(){return i;} /**<@return the number
                                  the value of i */
   private:
      unsigned int i;


   /** @var i
      @brief, you can put the comments where you want
                           with the special tags */
};
```

To update the Documentation of Tulip, you just have to use the makefile and so write : `make docs`.

9.4.3. Doxygen FAQ

9.4.3.1. How to insert a block of code ?

To illustrate your documentation, you can insert a block of code in a description between `\code` and `\endcode`. This code is written in the documentation with highlighting syntax.

9.4.3.2. How to force an end of line ?

Use the tag `\n`.

9.4.3.3. How to make doxygen ignore code fragment ?

It exists two ways to resolve this questions. The first one is to use the tags `\cond` and `\endcond` to skip the internal code. The second way is to use the preprocessor of Doxygen. In the configuration file, you specify the macros and you verify if the value of `PREPROCESSING` is to yes. Then, you set `MACROS_NAME` to `PREDEFINED`.

```
#ifndef MACROS_NAME

 /* code that must be skipped by Doxygen */

#endif /* MACROS_NAME */
```

Two macros are defines for *Tulip* documentations.

- `DOXYGEN_NOTFOR_DEVEL` : use to skip the code for Developer and User documentations.
- `DOXYGEN_NOTFOR_USER` : use just for the User documentation.

9.4.3.4. How to insert a equation ?

See the Doxygen web site[8]

Doxygen allows you to put Latex formulas in the ouput (just for HTML and latex format). Three ways are avaible. If you want to include formulas in-text, you have to put formulas between a pair of "\f$"

```
\f$ AB = \sqrt{BC^2 + CA^2} \f$
```

The second way is for a centered display on a seperate line. These formulas should be put between `\f[` and `\f]` commands.

The third way is to used formulas or other latex elements that are not in a math environment. It can be specified using `\f{`*environment*`}`, where *environment* is the latex environment, the corresponding end commands is `\f}`

9.5. References

Docbook XSL : the complete Guide http://www.sagehill.net/docbookxsl/index.html

The Duck Book - Docbook : the Definitive Guide [9]

The KDE documentation primer : recommandations to write correctly in Docbook - http://i18n.kde.org/doc/doc-primer/index.html

Doxygen web site - http://www.docbook.org/tdg/en/html/part2.html [10]

[8] http://www.stack.nl/~dimitri/doxygen/formulas.html
[9] http://www.oreilly.com/catalog/docbook/chapter/book/docbook.html
[10] http://www.docbook.org/tdg/en/html/part2.html