

# triAna

## An image triangulation visualization and analysis tool

Final report  
version 1.1

A bachelor project in cooperation with  
University of Chile  
Delft University of Technology

by  
Thomas Schaap (#1150561)  
Bruno Scheele (#1150588)

Supervisors  
prof. dr. N. Hitschfeld Kahler (University of Chile)  
drs. P.R. van Nieuwenhuizen (Delft University of Technology)  
ir. M. Sepers (Delft University of Technology)

## Preface

At the Delft University of Technology, students finish their bachelor studies of Computer Science after completing their bachelor project, where they are able to demonstrate all the skills learned in the previous three years of study.

For our bachelor project, the University of Chile has graciously accepted us to perform an internship for prof. dr. N. Hitschfeld Kahler. Currently, she is researching a way to speed up image analysis, by employing pre-generated triangulations in conjunction with the appropriate image, instead of merely performing pixel calculations.

We spent a total of 10 weeks in the care of the University of Chile and prof. dr. N. Hitschfeld Kahler, creating a plugin-based visualization and analysis program. This program will enable her to reimplement her triangulation algorithm and create analyses of images more smoothly and with direct, visual result.

We would like to thank both universities and prof. dr. N. Hitschfeld for the opportunity to study abroad, improve our skills as computer science technicians and for their time in assisting us to realise this project. We've learned a lot from this internship, both culturally and academically. We'd also like to thank drs. P.R. van Nieuwenhuizen for his time and guidance during our project.

Delft, September 2007

Thomas Schaap  
Bruno Scheele

# Table of Contents

Preface.....	2
Summary.....	4
Introduction.....	5
Progress report.....	6
Analysis.....	8
User analysis.....	8
Functionality.....	9
Design.....	11
General design.....	11
Design of the API.....	12
User interface design.....	14
Framework.....	16
Implementation.....	17
Working environment.....	17
Progress.....	17
Final results.....	19
Requirements.....	19
Original plan.....	19
Extra functionality.....	20
Documentation.....	20
Evaluation.....	22
Working procedure.....	22
Requirements.....	22
Planning.....	22
Design.....	22
Implementation.....	23
Documentation and testing.....	23
Working together.....	24
Working environment.....	24
Conclusion.....	25
Recommendations to the universities.....	25
Glossary.....	26
Appendix A: User's Manual.....	27
Appendix B: Developer's Manual.....	28
Appendix C: Examples' documentation.....	29
Appendix D: The UML-model of triAna.....	30
Appendix E: XML format for save files.....	31

## Summary

In this section, we'll give a summary of the report, reporting shortly what will be discussed in each section in more detail.

First we give an introduction of our project, triAna. After that we give a progress report, detailing our progress over the timeline and giving reasons for certain delays or changes in our planning.

The Analysis section shows the results of our user analysis. Four types of users have been found that will use triAna, namely researchers, programmers, teachers and students. To accommodate the needs of these user groups, we set up our functional requirements in which the main points are opening and showing images, their triangulations and performing analysis on those triangulations with visual feedback. triAna also has to be easily extensible and should be written properly according to good programming practices.

While designing triAna, we primarily tackled the extensibility and designed the plugin structure. For this structure, we had to decide what which amount of coupling we wanted for triAna. Because triAna had to be easily extensible, this gave some problems considering what type of user interface we should provide, but in the end we chose to let extensions provide their own parts for the user interface and to let the backend provide as much information as possible to extensions.

We designed the API with the above in mind. The handling of plugins, images and triangulations would all be done with singleton classes which managed them. Furthermore we decided to define the geometry in triAna and to not make this generic, meaning it can't be changed dynamically, since that wouldn't be useful for the user and create unnecessary complexity.

The user interface was designed in such a way that the user would be able to decide how the interface would look. All windows can be rearranged as the user himself desires.

The implementation of triAna was only done for Linux, as support for Windows was too difficult. The programming was done using Qt 4 libraries, which supported plugins, events and a simple way to implement powerful user interfaces. We did a fair amount of unit testing during the implementation, which helped significantly in finding and fixing bugs before they proved to be a big problem.

We've tried to resolve an issue with the slow rendering of large triangulations by adding a feature that lets the user himself decide when to update the view. This did not fix the issue, but it was a good workaround.

After this, we optimized certain algorithms to speed up the analysis and we've improved existing user interfaces and other features.

At the end of the project, we've completed all the requirements prof. dr. N. Hitschfeld Kahler had set for triAna. Although we couldn't implement some of the features we initially planned for triAna, we did implement several extra features, which improved the functionality significantly.

We've written documentation for the code, manuals for both the users and developers and written examples that the developers can use to implement new plugins for triAna.

In hindsight we have seen that several things have gone slightly awry: the planning of the testing phase was wrong, some design details should have been resolved much earlier and it took us quite some time until we'd figured out how to exactly make sure we were both thinking along the same lines. In the end these issues have not withheld us from reaching our goal and we consider the project to be a success. We've delivered a well working version of triAna which will enable prof. N. Hitschfeld Kahler to improve her research significantly.

# Introduction

At the University of Chile, prof. dr. N. Hitschfeld Kahler is conducting research on the automatic generation of meshes in both 2D- and 3D-environments based on images or 3D-models.

At the moment she is researching the analysis of images of tree stems by generating 2D meshes of triangles (triangulations) over the image and retrieving data based on these triangulations and the image, for example the amount of tree rings and specific data about the climate in the region from where the tree originated. The analyses performed using this method could be faster than performing them using traditional pixel calculations.

However, the current tools that are available for research are inadequate in three aspects. First of all, they're simplistic, non-intuitive tools that aren't designed for useful interaction but only create a single, simple output. They lack any kind of advanced functionality that could help speed up the research. Secondly, the program has little extendability and reusability, meaning it's difficult to improve the functionality and to modify it for different research within the same domain. And lastly, the triangulation algorithm and tools were produced as research for the Integrated Systems Laboratory (IIS), ETH Zürich, who retain ownership of them. This means that prof. N. Hitschfeld Kahler currently doesn't have the required permissions to continue her research with the current tools.

Our job was to create a visual tool which solved the previously mentioned weaknesses and which would enable an easy way to integrate a new triangulation algorithm and implement new tools to analyse the triangulations in conjunction with the images. We've named the program 'triAna', short for 'triangulation analysis'.

This report will discuss the process and results of our project. First we'll give a progress report of the entire project. The analysis of the requirements and the design of triAna will be discussed in the consequent chapters. The implementation and its problems and solutions will be discussed thereafter.

Finally we'll discuss the results of the project and do an evaluation. Afterwards, we'll offer some closing remarks and a few recommendations to the universities.

## Progress report

This bachelor project has been done at the University of Chile. At this university prof. dr. N. Hitschfeld Kahler researches the uses of triangulations of images. She did not have a good way to analyse them and during our first contact it became evident we could help her out by creating such a way.

The first week at the University of Chile has mostly been used to get a clear view of the requirements for a program that would enable such analysis. During this week we have not only talked to prof. dr. N. Hitschfeld Kahler to get a good idea of what she wanted, but we've also been brainstorming how to complement her requests to a complete set of requirements.

With the requirements complete, we've spent a couple of weeks designing what was to become triAna. Many choices needed to be taken and quite some questions still required answers. This has been the case until near the end of the project. Especially the way of representing complex selections turned out to be very challenging up to the very last minute.

One of the first challenges in the design was found when thinking about the connection between the user interface and the backend. Several options were available, allowing more and less flexibility and, conversely, usability. A choice was made to allow plugins to dictate a small part of the user interface, but to have the interface mostly dictate its own presentation.

A lot of time went into designing the backend of triAna. It is really the core of the complete program, in terms of actual functionality, and is used as a library. Hence good consideration was needed in its design. No big hurdles have been found here, although the final API has not been made stable until a few weeks before the end of the project.

Another part of the design was the visual design: how will the user interface look like? As stated before, this turned out to be a real challenge. A lot of things were rather straightforward and, researching a little for best practices and some usability studies, several were resolved in a very useful way. A few questions, however, have remained for a long time.

With the initial design laid out, the implementation was started. Several components of the system had to be made. The backend was done very quickly in a first version, but has been developed further throughout the project. During the development of the backend, testing was immediately done to ensure the quality of the backend. The user interface slowly marched on, requiring more and more complex parts. Two new Qt-widgets have been created for use with the interface, one of which had to be thrown away when the idea it was based on was thrown away as well.

Plugin-development has been started nearly at the same moment the backend supported them. A lot of plugins have been added in a late stage, since other parts had priority before that time and the API for the plugins was not completely stable yet. And still, even when all plugins had been made, changes were once again needed.

During the development several design issues and some requirement issues came up again. The plugin structure turned out to require too much hassle and several times the API, as it was designed, turned out not to be sufficient. Especially the changes to the actual API have changed the final design a lot, although all principal choices concerning the backend have been retained. The requirements have caused friction with the implementation at a few occasions, but none have not been implemented because of this.

Towards the end more and more testing and refining of code was done, alongside the further development of the code. Small details, such as installation instructions and licenses, have been added during the last few weeks, creating a complete product that is ready for use by other people.

It turned out to be impossible for prof. dr. N. Hitschfeld Kahler to use the product before the end of our stay in Chile, so no direct feedback was incorporated into the program. Many tests have been done to ensure good functioning of the code and while testing systematically many bugs have been found and removed.

We have always tried to adhere to the original planning of the project but due to mixing of some stages, such as early testing, this planning has not been kept entirely. A few things that were supposed to be done at the end turned out to be too much. The most important of these is the concurrent visualization of multiple selections. One of the causes of a lack of time for these is the time-intensive creation of a Qt-widget that was thrown away. Another cause is the long doubt over representing and using complex selections. Had this been resolved earlier a lot of unnecessary work and thinking could have been avoided and some extra work might have been possible.

# Analysis

This section describes the analysis done to gain all the information needed for the design of triAna. It will start with a user analysis, analyzing the different types of groups that will make use of the program and a short description of the main goals that had to be met. Afterwards we will describe the necessary functionality and show the initial list of requirements we made.

## User analysis

After our first meeting with prof. dr. N. Hitschfeld Kahler, we established the following four user groups that will make use of triAna. First we'll give a short description of the user groups.

### Researchers

Main user group, use triAna to conduct their research on analysing generated meshes.

### Programmers

Add functionality after release of triAna.

### Teachers

Use triAna to educate students in the creation of proper programs.

### Students

Will use triAna for their education. Will overlap with Teachers group and may overlap with Programmers group.

Each of the user groups will have specific needs that have to be met by triAna, as listed below.

**Researchers** will use triAna as a generating and analysis tool. Therefore they need to:

- be able to import graphics
- generate meshes based on the imported graphics and (user-inputted) parameters
- analyse meshes by using provided functions
- generate output based on their analysis

**Programmers** will want to add functionality to triAna, preferably without changing (a lot) of the source code. Therefore triAna should be written with the following goals in mind:

- all functions should be as extensible as possible
- the extra functions should be dynamically loadable, so the main source code doesn't have to be changed
- the code should adhere to proper programming practices (neat code, useful comments, including proper documentation)

**Teachers** try to educate students, unwilling as they may be. Because triAna will function within university context, it can also be used to show students a practical example of a completed program. Therefore the following should apply to triAna:

- it has to be coded using proper programming practices (neat code, useful comments, including proper documentation)
- the user interface should be easy to understand and to use

**Students** study computer science or something relevant and therefore will encounter triAna as a practical example of a completed program, usually because the teacher shows it to them. Because the students will have to study the code and practice writing computer programs using triAna as an example, the same goals should be met as in the programmers and the teachers group.



## Functionality

Taking into account the previous analysis, we had to make sure that triAna provided the functionality described below.

The main function of triAna is the analysis of an image and triangulation pair. TriAna should accept two files as input: an image and a grid-file containing the triangulation. Both file-formats had already been provided. The user had to be able to open the files, view them and make selections of triangles based on certain criteria. Viewing the files would result in an overlaid view in which both the triangulation and image are visible. The selections will be used to analyse the triangulations. An example of this is drawing all triangles with a very small area in purple. The criteria to select the triangles will be chosen by the user.

It would also be possible to open only an image and run the triangulation algorithm, provided by prof. dr. N. Hitschfeld Kahler, to generate an appropriate triangulation. The algorithm would operate automatically on the image and only output the resulting triangulation to triAna.

Afterwards the user would be able to analyse the triangulation and create specific forms of output based on the analysis.

Examples of user input for the analysis are selecting all triangles that have an area bigger than 4.8 pixels, selecting all triangles that have an edge with length bigger than 4.8, coloring all triangles according to the average color value of the pixels in a triangle, etc. Examples for the analysis based outputs are creating polylines around the selected triangles or drawing the selected triangles in purple.

triAna also had to be easily extensible, preferably without changing much of the source code. The program should ideally be extensible with plugins which require no changing of the source code at all. This required the program to be written dynamically and with proper documentation to facilitate these extensions.

Examples of extensions that were planned were support for more image formats in addition to the one currently employed by prof. dr. N. Hitschfeld Kahler. There was also a need for more advanced analysis and output extensions.

The goals that have to be met by the Teachers and Students groups are easily achieved. They require that triAna is coded properly and understandable.

Below is the point-wise list of all the functionality we initially had set as required for triAna:

- ◆ Loading and viewing of files
  - Loading and viewing images
    - Searching automatically for currently supported image formats
    - Support for Portable Graymap (.pgm) format
  - Loading any triangulation grid file and viewing it as an overlay over the current image
    - Searching automatically for currently supported triangulation formats
    - Support for .tri format
- ◆ Generating triangulations
  - Generate a triangulation based on a loaded image with prof. dr. N. Hitschfeld Kahler's algorithm using default settings
- ◆ Creating selections of triangles
  - Select triangles based on area
  - Select triangles based on edge length
  - Select triangles based on both area and edge length
  - Being able to save and load selections

- ◆ Outputting results
  - Visual representation of the selections.
    - Loaded image is background
    - Triangulation is overlayed upon image
    - Selections are overlayed upon both image and triangulation
    - Visualizing multiple selections at once
  - Saving and loading outputs
- ◆ Extensibility
  - It will be possible to add extensions to triAna without changing the source code:
    - For loading more image formats
    - For loading more triangulation formats
    - For making selections based on different user-criteria
    - For outputting the images, triangulations and selections
      - To other types of files
      - In other ways to the viewer in triAna

# Design

This section will describe the design of triAna. The design decisions have been based upon the requirement analysis, further conversations with prof. dr. N. Hitschfeld Kahler and brainstorming sessions. This section documents all such decisions and their alternatives as well as the rationale for the chosen alternative.

## **General design**

The first step in the design was looking at the program from a bird's view. How would the program be structured? How modular or monolithic would it be? Where did other parties have to be able to hook in? There weren't many questions to be answered, since the requirements told us a lot already. Nonetheless some doubts remained.

First of all the points where other parties would be able to hook into the program. The requirements already told us a plugin structure was required for image- and triangulation formats, selection-criteria and output based on those selections. So the end-user had to be able to tweak quite a substantial part of the program.

During one of our conversations, prof. dr. N. Hitschfeld Kahler gave the example of eventually providing functionality to automatically recognise specific features, like tree rings from photos of trees, to companies in order for them to work with the data. So apparently some form of automation (batch processing) could be needed later in the lifecycle of triAna. This left us with the question whether this should be taken into account and how we would take it into account. The general alternatives concerning automation are:

1. Don't take automation into account;
2. Take automation into account by allowing it to take over the user interface;
3. Take automation into account by allowing it to give parameters the user interface can automatically use to do the task;
4. Take automation into account by allowing it to give it's own interface or let it remove interfaces completely.

Looking at these alternatives questions about how the user interface interacts with the actual functionality of the program already came to mind. Since these questions were also important, and more pressing at that time, we took a look at those first.

The interaction between user interface and actual functionality has historically been done in a couple of different ways:

1. The user interface holds all functionality;
2. The user interface knows about all functionality, but the functionality itself is encapsulated in different classes;
3. The user interface knows about functionality and the encapsulation of functionality and can query the encapsulating classes for more information about available functionality;
4. The user interface is dictated by the functionality itself.

The most important issue that played here is coupling. The alternatives are actually ordered by their amount of coupling: alternative 1 has a very high coupling, while making it very easy to adapt user interface and functionality to each other, and alternative 4 has very low coupling, while making it very hard for a user interface to control the way the user works, since it's being dictated by the functionality (i.e. the functionality provides most parts of the interface).

High extensibility was one of the major requirements for triAna. This meant that it wasn't possible

to determine beforehand how the user interface would look and what kind of components would be needed at a time for a particular user. This became most clear when thinking about visualisations and criteria: criteria will need parameters to be set and will thus want to interact with the user and visualisations might want to show many useful statistics to the user, such as diagrams showing color-distribution within a selection or statistical analysis of a selection. Such extensions that wish to interact closely with the user can be very useful, but also require the interface to either know about them beforehand or allow them to change the user interface. Since we did not know what extensions would be available by the end of the project and beyond that, we couldn't incorporate them directly into the user interface. We needed some way to allow extensions to influence the user interface. Alternatives 1 and 2 thus became impossible: developers don't know about all functionality, let alone the interface they'll develop.

Another requirement that has been mentioned during the requirement analysis was usability. Usability means building a good user interface that's understandable and intuitive for the user. The most perfect situation for this to be possible is when all functionality is known to the developer of the user interface: he can then make the best choices for the interface. This collides head on with the need to have an extensible interface, though.

Since both requirements, being extensible and being usable, are definitely there a path in between had to be found. The best way seemed to be to let the user interface know as much about the structure of the functionality as possible and allow the extensions to tell the interface how to display specific parts.

Having decided on this decoupling of interface and functionality, we could rethink our question about automation. Since the interface and functionality would be set apart, automation shouldn't have to be too difficult: we could simply replace the interactive user interface with an automated interface and do whatever is necessary to do; the functionality won't really know the difference unless it's truly dependent on the interaction with the user.

Allowing the user interface to be replaced by an arbitrary other interface (be it interactive or automated) also meant the program could be used by many different types of users by simply building a different interface for them. This means that triAna won't need to remain a tool purely for science, but can also be used by others to actually do some work.

An added advantage to this approach would be an inherent increase in quality of the functionality: since extensions are now forced to interact in an interface-independent manner (of course excluding such visualisations as diagrams, who really need an interface) they have to be well thought out and will hence behave and function better, adding to the usefulness of triAna.

Allowing automation by allowing the interface to be replaced seemed like the best idea. It did call for a way to let the interface and actual functionality communicate. This could be made possible by allowing the interface to talk to the different components of the functionality, effectively creating an API for the interface to use.

## ***Design of the API***

The API of triAna holds the core functionality. It is the point of entry for external parties to add their own parts to triAna. This put some heavy requirements on the API's shoulders. Most importantly the API needed to keep track of everything that's connected to it in order to use it. It would also provide the functionality of all internal and extensible functions in a uniform way to those who need it.

At its core, the API would need to be able to handle all extensions to triAna. This concerned all extensions to image formats, triangulation formats, selection-criteria and outputs. There wasn't much doubt over how to do this: plugins traditionally provide an excellent extension scheme for applications. Considering memory-management and administration, managing the plugins is a task

that should be done centrally. Hence it was only logical to make the manager a singleton.

Searching for the available image and triangulation formats was another task for the API, closely related to plugin management. Two singleton classes would handle these tasks. They would be different classes, since their functions are distinct enough from managing plugins and distinct enough from each other not to put them into the same class. They should be singletons since searching for an available plugin is a task that only uses the manager of plugins and the plugins themselves. It makes absolutely no sense to have more than one of these around, even when having more than one instance of triAna opened: the added functionality when one user loads a plugin is only a free bonus to the other user.

The functionality of triAna is all about triangulations. Hence it's rather useful to have a way to describe the geometry of such a triangulation. It was only logical to have the API take care of this. A consideration we made is the mutability of such a geometry. It would have been nice for some extensions to be able to change the geometry. Then again, that would add a lot of complexity to the geometry and could potentially confuse future extensions, up to the dangerous level, even, where one changes the geometry because another changed it, who will change it because the other changed it, etcetera in a vicious cycle. Taking into consideration that analysing the geometry usually only is useful on the initially generated geometry, we couldn't think of any use-cases where changing the geometry is necessary. Therefore we decided to create the geometry in an immutable way and to disallow any changes to take place once the geometry has been created.

The interface-independence of the API meant the geometry can't hold any data concerning its representation (presentation of vertices, the thickness of edges, the color of triangles, etc.) This actually held a challenge, since that would mean that only the interface knows about representation, which in turn meant that extensions wishing to change the representation would need to know about the interface, which is impossible due to the interfaces interchangeable nature. This challenge could have been solved in different ways:

1. Dropping the decoupling of interface and API;
2. Dictating the number of ways an extension can influence appearance of geometry;
3. Disallowing extensions to influence appearance of geometry.

It's rather apparent from this list which approach we chose for our design. Alternative 1 goes straight against the previous decision to actually decouple interface and API. Alternative 3 makes one of the requirements (letting extensions alter the visual output of triangulations) impossible to fulfill. Alternative 2 seems very restrictive, but isn't so that much. Adding many ways of influencing appearance is useless, since the user wouldn't be able to distinguish between the different appearances if there is too much detail. Also, there are only a limited number of ways in which a single part of a triangle can be displayed with any significant difference.

An important addition to dictating the ways in which geometry appearance can be influenced was dictating that interfaces are free to choose whether to actually use the representation. This allows, for example, automated interfaces to ignore such effects and interfaces for color-blind to use a different means of visualising color.

The logical location for this type of data to be placed is in the geometry itself. One might argue that combining data and its representation is a bad thing, but not only are these representation properties abstracted, a separation of the data and representation will result in the representation being held in almost the same way as the data: divided in points, edges and triangles. Hence separation would give a lot of overhead, while the usual benefits of separation, ease of change, is absent. This lead us to believe it's best to hold the abstracted representation properties in the geometry itself.

At the core of the work of a user of triAna is the analysis of a triangulation. Analysing a

triangulation will mean combining selection-criteria and generating some output based on them. To enable this in the API we built a combination class, which allows the interface to easily combine selection criteria and output and to operate on them. Such an object would make building an interface much easier, which would translate in decreased complexity and code duplication in the interfaces.

Earlier it was decided to decouple the interface and API, but to allow extensions to influence the interface to interact with the user. The way this would be done is by allowing extensions to provide the interface with a widget that can be displayed to the user. In order to allow automated and other different interfaces to function, the restriction has been made that every option and output should be made available in a programmatic way as well, so the user interface can query and change said options without the need for a visual interface or user interference.

We've created a UML-model of the classes in the backend on which we based our initial implementation. During implementation, we came across several design issues, which required the programming of additional classes or methods. We've kept the model updated during these changes. The model in appendix D is designed to be read with the documentation, detailing only the classes, their relations and their public methods. The exception to the latter are the protected methods of the plugin interfaces, which are also listed.

## ***User interface design***

Despite allowing different interfaces to be used with triAna, we were to build an interface ourselves which would allow for interactive use of the application to actively do research on triangulations.

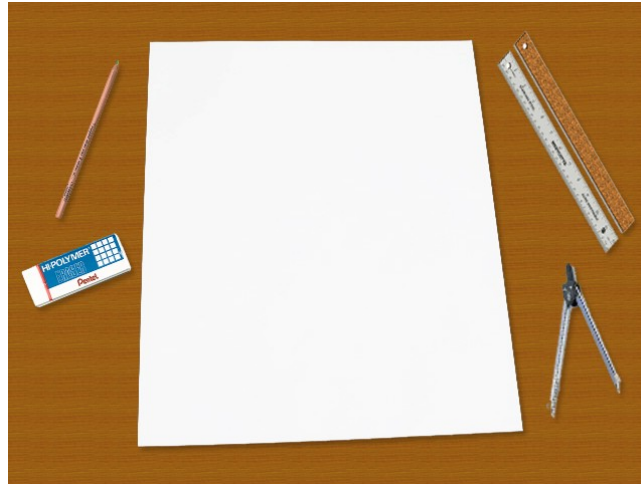
While we wanted users that are unfamiliar with triAna to be able to work with it, we did expect all different user groups expected to work with triAna (previously mentioned in the analysis section) to have basic knowledge about the research and its terminology and to have a basic knowledge about computer programs as well. It wouldn't be beneficial for the research to base the user interface on the assumption that a user who is only vaguely familiar with the research and has little to no experience with computers to have to be able to work with triAna.

Experience also points out that users have a lot of different ideas about the proper layout of a program. This makes it difficult to effectively design a layout that is consistent with the needs of most individual users. Users want to be able to customize their program to their own needs.

Under these assumptions, we defined the goals for triAna's user interface to be as follows:

- The user interface has to be relatively easy to understand and use for users familiar with the triangulation research;
- The user interface has to be customizable to adapt to the needs of different users.

To reach these goals, we decided to model the user interface with a desk metaphor (as shown in illustration 1). This means that the user will work on a desk, with the main worksheet of his focus (the picture to be triangulated) in the center of his workspace and all his tools (the extensions that do the analysis) layed out around the worksheet in a way that the user prefers.



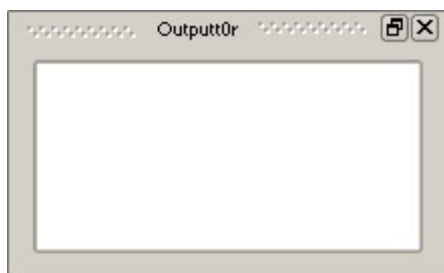
*Figure 1: Desk Metaphor*



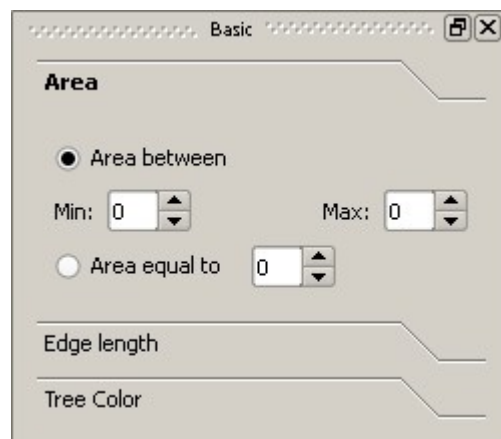
Thus, we designed triAna around a main worksheet (illustration 2) that will contain the image to be analysed. All the needed extensions would have their own toolbox which contains all their functionality. These toolboxes (illustrations 3a and 3b) could be moved around the screen independently and positioned any way the user wants.

It had to be possible to add and remove loaded toolboxes as needed and to add new toolboxes, should the need arise.

Any user configuration could be saved and loaded, to prevent the user from having to setup the application every time when he restarts it.



*Figure 3a: An example of an extension in a toolbox*



*Figure 3b: An example of an extension in a toolbox*

Aside from user configurations of the layout being saved, it also had to be possible to save sessions. Sessions are defined as the current state of the program, meaning a user can save and load all progress made during one session.

## **Framework**

In the design several issues have not been thought out. One example of this is the use of plugins. Plugins can be rather hard to implement, especially if they're to be loaded dynamically. Another thing that has not been mentioned, but has been showed, is the widget framework used for triAna. We decided to use Qt 4 shortly after the first conversations with prof. dr. N. Hirschfeld Kahler. Thomas already had experience with the framework and prof. dr. N. Hirschfeld Kahler was also familiar with it. It has a lot of functionality incorporated that we needed, such as plugin management, widgets and events. It also provides complete platform-independence.

*Figure 4: Example of a saved layout*



## Implementation

During the implementation phase the code was written that constitutes triAna. During this phase several steps were taken ahead, such as testing and documentation, especially since the code was almost directly used again in other parts of the system. Hence the need for tested code and documentation became evident rather early.

Not all of the design turned out to be completely correct, which became a hurdle at times. The most sensible solution at these times was to simply change the design. Therefore the original design is no longer the actual design that has been implemented. The general structure has remained the same, though.

## *Working environment*

We used our own laptops for development of the code. We were offered computers from the university, but using our own systems seemed the better idea, since we could easily build a development environment on it and test as much as we wanted to. At first we used a dual environment, partly Microsoft Windows and partly Linux, but since support for Microsoft Windows turned out to cost a lot of time and was not required, we switched to Linux only after a few weeks of development.

To keep work ordered and make working together more easy, we've used subversion for the source-code, keeping revisions and merging changes when changing the same file. The last function was especially useful for the large files of the user interface.

## *Progress*

The first that had to be done was creating a working backend. The backend is the part of triAna that does the actual calculations on triangulations. These calculations were to be placed in plugins, which would be loaded and used by the backend. After setting up the build environment, consisting of CMake with gcc and the Qt 4 libraries, the backend was implemented in its first version within a week. Some tests had already emerged by that time, especially testing generic plugin functionality and the geometry classes.

With the first version of the backend in place, development of the plugins, and especially the input plugins, was started while work also began on creating an interface. A widget was developed to contain the plugins in a useable way. This widget turned out to be both a failure and not a useable way at all. This is due to the questions about using and showing complex selections, which were still not resolved by this time. Alas it had also cost a lot of time, mostly because it was derived from another Qt widget, which is declared privately for more than 50 percent, making it impossible to simply extend it. Integration of such highly customized widgets into the user interface builder and compiler of Qt turned out to be very difficult as well.

With the backend slowly improving over time some problems were encountered in the first version and partly also in the design. Especially the structure of the plugins has had a few revamps: missing functionality, misplaced responsibilities and incorrect object-management were the most common errors.

Nearly three weeks into development the created widget was added to the user interface. Initially this seemed to be a success and at least development could go on. In the meantime plugins kept emerging and improving. Also, since it was being used more and more, the backend received more

and more bugfixes in the complex classes, which had not have been fully tested before.

By the time displaying images and triangulations was working well a new widget was added to try and solve the problems with rendering: this was way too slow. Using the Qt classes for this was definitely the nicest way to do it, but the Qt visualization class (QGraphicsView) could not handle the tens of thousands of small visible objects very well. It became very slow, using more than ten seconds to draw its contents when stressed well enough. Luckily high speed was not a requirement and preference could be given to nice coding. But some way to improve the speed was still required: resizing or dragging windows had become impossible once a big triangulation was loaded. The new widget fixes the issue by only redrawing when ordered to.

After integrating the new widget many time has been spent on optimising the plugins, creating new plugins, support for saving an analysis, which required some small changes, and optimizing the user interface. The latter was difficult at times, since strange things came into existence because of the failing way of representing the selections. The greatest abomination was probably being the creation of empty widgets just to be able to show the plugin in the user interface. Such oddities triggered the rethinking of the interface. By then so much time had been spent on the concept that the final way of representing the selections was thought of and could be put into place.

The last weeks have been spent on improving the interface, plugins and documenting everything to meet the high standard we required. During this time support has also been built for groups and subgroups of selections to allow arbitrary complex calculations.

## Final results

After the implementation phase the project has been rounded up and, with the final details of licensing and installation instructions, made a first final version. Nearly all requirements have been implemented in this version and extra functionality has been added as well.

## Requirements

Most requirements from the first analysis have been implemented in the current version of triAna. The requirements that have not been met are discussed below.

### *Generating triangulations*

Originally it was planned to have triAna generate a triangulation of a loaded image using prof. dr. N. Hitschfeld Kahler's tools. This has not been fully implemented, since the tool used to generate the triangulations is propriatry and could not be used in conjunction with triAna. This was no problem, though, and support has been implemented for plugins to provide similar functionality. An example of this is added to the source: the TriangulationGenerator plugin from the examples takes all steps up to the point where the actual calls to an external utility would be needed. Lacking such tools the example asks the user nicely to create the triangulation by hand.

### *Visualizing multiple selections at once*

This turned out to be a difficult thing to implement in the user interface. The backend does support this, but the interface can't show it, currently. The main problem here is that to maintain the usability, we want the main screen as free as possible from clutter and an abundance of controls. It was already difficult to design a workable user interface for a single selection. Visualising multiple selections, though possible, would increase the clutter, making triAna less useable. Given more time this might be doable, but serious user interface design is needed to create a useful way for doing this.

### *Saving and loading outputs*

Although saving and loading has been implemented, saving and loading outputs is not implemented as intended when first thought of. The idea back then was to take the output of a selection, combined with its triangulation and image, and save that. However, loading this had not been thought about well. When we did so, we reached the conclusion that loading only the results is a rather useless feature. This requirement has then been changed to saving and loading the analysis as it is being conducted, effectively saving what plugins have been used with which settings, in what order and how they were grouped. This implements the requirement as far as saving and loading in triAna is concerned. A specific XML format has been devised for this, which is included in appendix E. Saving to another format has not been implemented directly, but is certainly supported. A direct example of this has not been provided, but it can easily be implemented by a plugin. It has been documented in the developer documentation in the chapter 'Plugins with actions'.

## Original plan

The original plan did not only include creating a tool to analyse triangulations, but also to analyse the treering examples provided by prof. dr. N. Hitschfeld Kahler. This has not been done completely, since we did not have the time to do so. One or two extra plugins will do the job, though, and the example analysis in the user manual actually is the analysis of the treerings example, creating a rather good selection of the treerings. One way to create a file with polylines from these might be to implement a plugin to deselect parts from the selection that are clearly not

part of a (broken) circle in that selection, implement a plugin to create complete polylines of what look like circles and a plugin to output those lines to a file.

## ***Extra functionality***

Along the way several extra functionalities have been added to triAna. These will be discussed below.

### *Analytical and statistical information from plugins*

The first of those functionalities was the ability to have information from plugins displayed. Plugins can provide information, such as the area of the smallest and biggest triangles in the selection, which can be displayed by a frontend, such as the standard user interface. The main intended use for this is analysis and statistics, but other information may be showed as well.

### *Viewing modes*

A small cosmetic point that turned out to be useful, nonetheless, is the ability to choose the mode of viewing: only the image, only the triangulation and selection, or both layered together.

### *Plugins*

In the final stage's analysis and design more plugins have been identified than anticipated at the beginning of the project or required by prof. dr. Hitschfeld Kahler. The reason for these extra plugins is the vision of complex analyses that can be built up using the set functions.

### *Set functions*

Primarily selections were thought of as a simple row of plugins, each creating a subselection of the previous's subselection. This way of thinking about selections is both complicated and very limiting. Therefore set functions were introduced, along with grouping of plugins. This allows for flexible creation of complicated calculations. An example of this can be found in the User's Manual in Appendix

### *Quick preview*

Another cosmetic, but highly useful, function is the quick preview of the output of a single plugin somewhere in the middle of the calculation. Using this function the visualization does not show the output of the complete selection, but the direct output of the selected plugin. This function is not supported by the backend, but is an added feature in the frontend, which partly duplicates some code from the backend to do this.

## ***Documentation***

The requirements of triAna also stated that it had to be user-friendly and easy to learn and also that developers should be able to easily write plugins for triAna to extend the functionality. Therefore it was paramount to document triAna properly in two ways.

We wrote a user's manual (included in appendix A), which details all the functionality in triAna that's currently available to the user and explains how it's used. Because the user is not the same as the developer we chose not to write about currently supported features that are not available through the user interface and features that are not in use.

For example, plugins may currently have actions and rightclicking on the appropriate plugin in the selection or output list brings up a context menu listing these actions. However, this wasn't part of the initial requirements and we implemented these to facilitate the developers. There are no actions in plugins yet and thus the feature is not detailed in the user's manual.

For the developers we wrote a developer's manual, which details the use of all parts of the backend's API and explains how to extend triAna using plugins or a new frontend. We've included the developer's manual in appendix B and because it refers often to our examples, we've included their source codes in appendix C. All implemented features are explained, including currently unused features (such as actions for plugins).

Furthermore, the developer's manual also gives examples for implementing plugins of all types and one small example for writing a new frontend. These examples show the major steps required to write a plugin, to ensure that the developers have a clear example to expound upon.

## **Evaluation**

Looking back at the project we can say more about how everything went. In general we have a good feeling about it: the project was ended with a good result that was received very well and along the road there have not been any major problems. A good look at different aspects will be a good lesson for the future. Below these evaluations can be found.

### ***Working procedure***

To complete this project, we started out with the usual phases of analysis, design and implementation of triAna. During implementation, we constantly checked if our current design was up to sufficient to satisfy all our requirement. If we found that we needed to rethink an aspect of our design, we went back to the analysis phase focused on the design aspect. There we decided what the best change would be and how it would affect the rest of the program. Given that we were a team of only two, this approach has worked well and allowed us to progress fast and remain flexible about the design.

### ***Requirements***

The requirements analysis consisted mainly of a few conversations with prof. dr. Hitschfeld Kahler and some brainstormsessions by ourselves to complement her requirements.

The requirements we got from prof. dr. N. Hitschfeld Kahler described a system with the basic functionality as found in triAna, with a useable interface. Although her requirements gave us a line to follow, they did not describe a complete system. Even though we would have liked her to give a clearer view of what she wanted, this was to be expected and made the requirements analysis a nice challenge.

Complementing the requirements consisted mainly of more extensibility, dynamic extensibility (as opposed to recompiling the source code), the selection of plugins and the final user interface.

### ***Planning***

The initial planning of the project turned out to be quite correct. Some time was lost along the way with uncertainties and unnecessary work, but overall the planning was fine.

One thing that should have been different was the initially planned last phase: the documentation and testing can't be done at the end, especially not when it concerns the backend that will be used by other parts of the system throughout the rest of the project. In hindsight we could have foreseen this: how can one build a system, use it extensively, and finally write some documentation about how to use that system? This became evident pretty soon, when tests and documentation were needed in the first week of implementing the backend.

### ***Design***

By and large, the design of triAna went well. We faced several choices during the design, which we thought about and where trade-offs were considered. The choices have been documented for future reference, which turned out to be useful later on.

Designing the backend was, as expected, a challenge and not completely correct in its first revision. Several changes have been made throughout the development process. The real difficulty here is that, until development using the backend starts, one can't have a completely good view of what

will be requested from the backend. Likewise, some properties and events seem to be useful at first, but turn out to be totally useless and simply overhead wherever they pop up.

The interface design of triAna posed one of the greatest challenges, both the evolving technical design and the usability design. The first visual version of the interface design was highly generic and had not worked out very well how plugins would fit in, would be used and how selections would exactly be created. Thinking more about this might have prevented many difficulties during the implementation.

## ***Implementation***

The implementation went relatively smooth, taking the earlier mistakes into account. Some problems were encountered with the actual environment and at times some more documentation reading might have prevented some wasted time, but this won't be much.

Taking the earlier mistakes into account, most time wasted here was on inexperience with the environments and the language. As said, this could have been prevented by reading more documentation, writing better and more extensive documentation ourselves and more experience with C++ and Qt. Lack of experience, however, was already known and some trouble with this was expected.

A real problem that temporarily rose was support for Microsoft Windows: this broke when we started using the backend as a library. Luckily we could drop this support, since it might have been a lot of work, and possibly frustration, to have that fixed.

## ***Documentation and testing***

Documentation and testing were at first planned as a different phase. This was definitely wrong, as stated before. Writing the documentation was not much of a nuisance, as is often a reason for bad documentation, especially since we knew why we wrote it: we needed it ourselves, which gives one a good appreciation for good documentation. Writing good documentation, however, is not as easy as it seems, and often we have had discussions about how things were supposed to work, since one had written the documentation and the other had interpreted, and used, it differently.

Testing systematically turned out to be a great asset to our productivity. We were amazed to find ourselves writing small tests and finding many bugs with it within virtually no time. Testing everything directly, as far as possible, has most likely prevented many hours of searching for bugs.

Testing was done with the testing framework of Qt, which easily allowed us to perform unit tests on each of the classes directly after these were completed. The unit tests consisted of creating a hand coded, exhaustive test for all the class' methods and checking to see if their outcomes corresponded with the desired outcome. The main benefit of this kind of testing is that small errors, mainly those that exist when applying boundary conditions, are revealed instantly after you finish a class and can be fixed before they become a problem later on, where it could be difficult to track them down. Another advantage was the possibility to easily test for regression: unit-tests are fully automated and running them after every build has prevented some errors that would have been introduced by changes in seemingly irrelevant classes.

While unit tests perform admirably for classes, sometimes faults occur that were not found on the unit tests. These kinds of faults mainly introduced themselves by crashing the program. These were sought out by running the relevant build of triAna through the debugger utility of KDE.

## ***Working together***

When working closely together for 10 weeks, some things are bound to come up. First and foremost, though, we can state it has been nice and productive to work together. One of the problems that was bound to come up is the difference in viewpoint. Bruno being a student in Media and Knowledge Engineering (MKE) and Thomas being a student Software Technology (ST), a difference of view was to be expected. In the beginning this has given some difficulties, especially when trying to work out issues with the user interface or with the API of the backend. During the project, though, we realized these differences existed and held us back. Thorough conversations have worked quite well to solve such differences, or at least to understand the other's point of view. It was in one of these conversations that the representation and use of complex selections was discussed and finally solved.

Another difference that was bound to come up was the difference in programming skills. Bruno had less experience in programming and using complex and abstract APIs. This has led to much of the technical details being done by Thomas. Luckily this was also anticipated and Bruno has done a lot in the interface design and improvements, considering Thomas has a lot less experience in those matters.

## ***Working environment***

Since we could work on our own laptops we were able to prevent a lot of problems that are concerned with system administrators. We had a few issues with access to the network, but the administrators of the University of Chile were very helpful in such cases.

At first the dual environment, Microsoft Windows on one laptop, Linux on the other, worked fine and did not give a lot of issues. The differences between the two became very apparent when in Linux a file was created, called README, and committed to the svn. A lot of work was needed to find out it needed to be renamed to readme, since the case-insensitive Windows-system didn't like the name README. Not many other issues arrived with this, though, until compilation started breaking and support for Microsoft Windows was finally dropped. Switching to Linux took some time, but once settled it was not a real problem and no more issues were found.



## **Conclusion**

We consider this bachelor project a success. We set out to Chile to help prof. dr. Hitschfeld Kahler with her research and when we had reached a first final version she was most happy with the result. The tool we created will help her both in her research, where she will be able to actively analyse triangulations in a useable and flexible program, and in her educational role as a professor, where she will be able to show the program to her students as an example of well written code and use of design patterns.

Not only will our bachelor project be a real asset to prof. dr. N. Hitschfeld Kahler and the University of Chile, we ourselves have greatly benefited from the experience as well. Being able to do a part of ones study abroad in a very different country with a very different culture, is already a great experience from which one can gain a lot of personal growth and pleasure. And beside that our knowledge and experience in our area of expertise has improved as well.

## ***Recommendations to the universities***

Our experience in Chile had been a great one and it is our opinion that we need not be the only ones to have this experience. The cooperation with the University of Chile was a very good one and Chile is a nice country to visit. Therefore we would recommend both unversities to exchange students on a more structural level. Here one can think about bachelor projects, as we have done, but also about following a part of the courses at the other university. We have met several people in Chile who did the latter and they had very good experiences with the education at the Chilean universities, although none of them studied computer science. We did take a look at the studying programs of both universities, though, and they seem to be virtually the same. We think exchanging students would be a great addition to both universities' international contacts.

## Glossary

analysis	The current image and/or triangulation with all the tools currently in use. Also refers to a single complete workstate.
backend	The core functionality of triAna able to function even without a user interface.
coupling	The dependency between different parts of computer programs, in this case between the backend of triAna and its supported plugins.
frontend	A user interface or automation designed for use with triAna.
image	An image that can be loaded into triAna if it's of the supported file types.
output	Any form of output of the current selection, be it visually, textual, numerical or all of the above.
plugin	An extension of triAna that can be loaded dynamically to increase the functionality concerning the loading of images or triangulations, selection criteria and forms of output.
Qt	Platform-independent development framework which provides, among other things, support for plugins, widgets and events. Used to program triAna.
selection	A selection of triangles, edges and vertices in a triangulation
tool	A triAna extension, creating a selection or output based on a selection.
triAna	An image triangulation and visualization tool, the main program and result of this bachelor project.
triangulation	A 2d-mesh of connected, non-overlapping triangles, based on a reference image.

**Appendix A: User's Manual**

# triAna

## An image triangulation visualization and analysis tool

User's Manual  
version 1.0

A bachelor project in cooperation with  
University of Chile  
Delft University of Technology

by  
Thomas Schaap (#1150561)  
Bruno Scheele (#1150588)

Supervisors  
prof. dr. N. Hitschfeld Kahler (University of Chili)  
drs. P.R. van Nieuwenhuizen (Delft University of Technology)  
ir. M. Sepers (Delft University of Technology)

## Table of Contents

Overview.....	3
Getting started.....	3
Main screen.....	4
Menu.....	5
View screen.....	6
Selection.....	6
Output .....	7
Information.....	7
Tools.....	8
Selection.....	8
Triangle selector by area.....	8
Triangle selector by edge length.....	8
Triangle selector by average pixel value.....	8
Edge selector by difference in area.....	9
Edge selector by difference in average pixel value.....	9
Vertex selector by triangle amount.....	9
Triangle remover.....	9
Edge remover.....	10
Vertex remover.....	10
Edges from triangle boundary.....	10
Internal vertices from triangles.....	10
Output.....	10
Color by average pixel value.....	10
Selection colorizer.....	10
Histogram of triangles by area.....	11
Histogram-based triangle colorizer by area.....	11
Histogram of triangles by average pixel value.....	11
Histogram-based triangle colorizer by average pixel value.....	11
Example.....	12

## Overview

triAna is a program designed to help the researcher in analyzing triangulations based on images. The ability to perform calculations on the triangulation and image pairs and visualize those, simplifies discovering new data for the researcher. The calculations can be performed on the triangles and the edges and vertices they consist of. Likewise a single selection can contain all these elements for use in further calculations.

triAna has the following features:

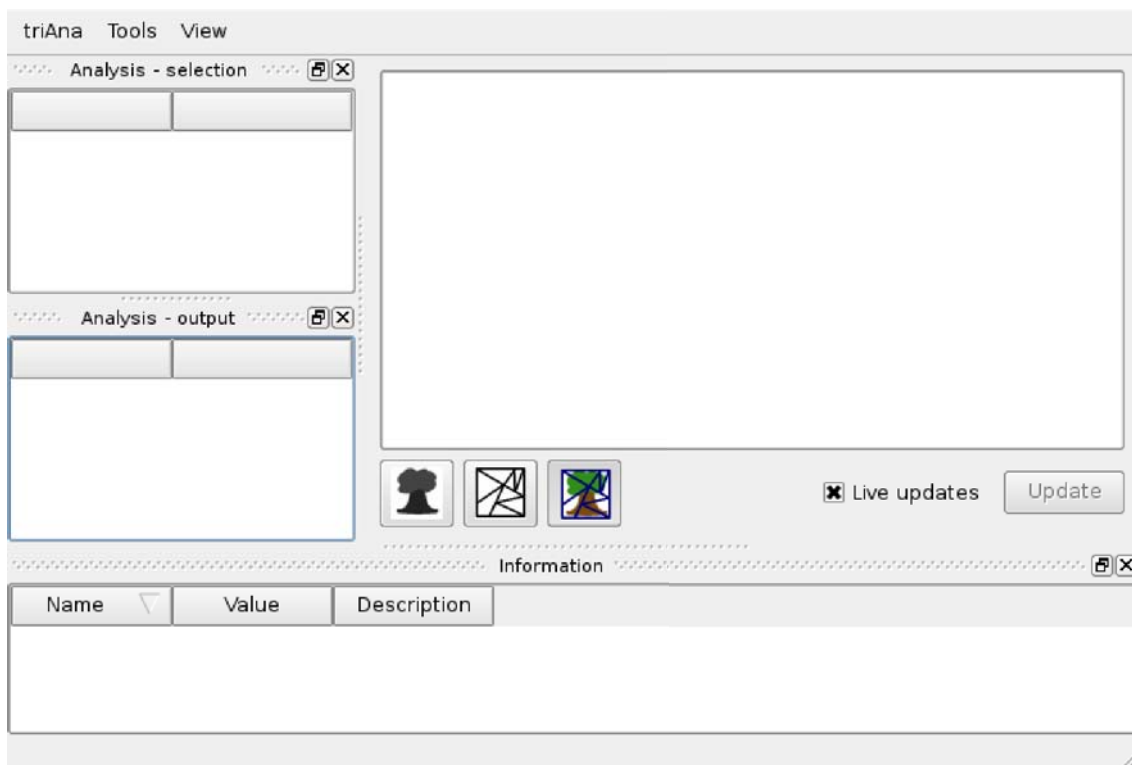
- ◆ Loading a variety of image types.
- ◆ Loading triangulations from TRI-files.
- ◆ Perform calculations on the triangulations and the triangulation-image pairs.
  - Perform multiple calculations in a user-specified order.
  - Create and combining sets of calculations using set operations.
- ◆ Visualize calculations
  - By coloring the triangulation.
  - Through histograms.
- ◆ Load additional plugins to extend functionality.

## Getting started

Getting started with triAna is easy. Merely copy the installation to your desired location and run triAna from the 'bin'-directory. Afterwards, the main window will open (illustration 1). triAna consists of the main screen with menu bar and the selection, output and information windows. The windows are currently docked to the sides of the main screen, to ensure that triAna runs in a singular window.

The position of each window can be changed by dragging it to another side of the main screen and under other windows. It's even possible to create tab windows, by dragging windows onto each other or to detach them completely from the screen, by dragging them to an empty spot in the screen.

This way it's possible to adjust triAna to your wishes. The windows will remember the position they were on closing, so starting triAna again will result in the window positioning itself where you last left it.



*Illustration 1: triAna after starting*

## **Main screen**

The main screen (illustration 2) is where all the calculations performed become visible in the view screen. It also holds the menu bar, where you can load an image or triangulation, save and load a workstate, use the tools available in triAna and remove ~~from~~ or restore windows on the screen.



*Illustration 2: Main screen*

## Menu

The menu (illustration 3) consists of the following items and their options:



*Illustration 3: The menu bar*




- **triAna**
  - *Load image*  
Loads images of the following types: BMP, GIF, JPG, JPEG, PNG, PBM, PGM, PPM, XBM, XPM
  - *Load triangulation*  
Loads triangulations of the TRI-format.
  - *Load workstate*  
Loads a previous triAna workstate, enabling you to continue your work where you last saved it.
  - *Save worstate*  
Saves a triAna workstate, so it can be continued at a later time.
- **Tools**
  - *Load plugin*  
Loads a plugin for triAna, adding functionality.
  - *Selections*  
Opens a side-menu from which you can select a tool to use for making a selection.
  - *Output*  
Opens a side-menu from which you can select a tool to visualize the current selection.
- **View**



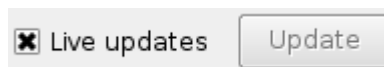
- *Information*  
Switches the information window on and off.
- *Analysis selection*  
Switches the selection window on and off.
- *Analysis output*  
Switches the output window on and off.

## View screen

After loading an image and a triangulation, the screen has the following options, displayed as buttons:

-  View only the image.
-  View only the triangulation and the output of the current selection.
-  View the image and triangulation together.

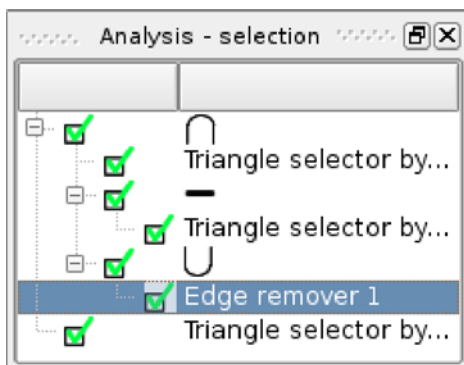
Because a typical triangulation can consist of a very large amount of triangles, refreshing the view constantly to show any changes can be very slow. That's why triAna gives you the option (illustration 4) of choosing whether you want the view to refresh automatically or refresh it manually.





*Illustration 4: Refresh options*

- **Live update**  
When checked, the view refreshes automatically on any change.
- **Update**  
When 'Live update' is unchecked, triAna creates a bitmap of the current output and displays that instead of a live rendering. This ensures that changing settings and resizing the view screen go smoothly, when handling large triangulations. Afterwards, you can choose to update this bitmap manually by pressing this button.

## Selection



*Illustration 5: Selection menu*






All the selection tools that you select from the Tools-menu (illustration 5) appear here for further management. A tool always operates on either triangles, edges or vertices and leaves the rest of the parts in a selection alone. Thus, if you start with a complete selection (with all the vertices, edges and triangles) and use a tool that selects certain triangles, you'll end up with a selection that contains a subset of triangles and the complete set of edges and vertices. Each tool is disabled by default, which means that it will not be included in making the current selection. You can enable (  ) and disable (  ) the tool by clicking the check box next

to it.

triAna processes all tools from above to below. It takes the selection, performs the calculation specified by the tool and returns the result for the next tool. To change the order in which tools are processed, you can drag tools around the menu and drop them in the desired location.


### Set groups

triAna has the capability to create groups of tools, which enable triAna to perform a set operation on the results. When the triAna process reaches this group, it takes the input of that group (the current selection) and uses that to process all the tools in the group, according to the function selected.

-  *Linear*  
Takes the input and processes it linearly with all the tools in the group. This is the default option.
-  *Union*  
Takes the input and processes it with all the tools in the group separately. After that, it returns the union of all the results.
-  *Intersection*  
Takes the input and processes it with all the tools in the group separately. After that, it returns the intersection of all the results.
-  *Complement*  
Takes the input and processes it with all the tools in the group separately. After that, it subtracts the results from the original input and returns that.
-  *Subtraction*  
Takes the input and processes it with the first tool it encounters. Afterwards, like complement, it processes that result with the remaining tools, subtracting all those results from the first result and returns that.

To create a group, just right-click on the selection menu. Then select 'Insert group' from the resulting context menu. It will then create a group that's set to the default set function. If you select the group and then select a tool from the 'Tools'-menu, then that tool will be added to the group. You can also drag already enabled tools to a set. Changing the function of the group merely requires you to select a function from the context-menu after right-clicking it.

### Displaying a single result

When you have a lot of results, it is useful at times to see what the result is of a single tool, without going through the trouble of disabling each and every tool below. That's why right-clicking on a tool also brings up 'Show result'. Checking this, replaces the current output in the main screen with the output that the selected tool would have. You can disable this by selecting 'Show result' from the context menu again or by clicking on the eye-icon  which replaced the check box.

### Output

An output tool colors the selection created by all the selection tools in the main view. Some output tools don't adjust the main view, but give a graphical representation in their own window (for example, creating an histogram).

The output window largely functions like the selection menu, though there is no possibility to create groups or to show the result of a single output tool. This means that the tools all adjust the main view consecutively and the final result is shown.

### Information

Some tools contain additional information that may be useful to you. This information is shown in the information window. Each entry has its own name, value and description.

## Tools

Here follows an explanation about all the tools currently available in triAna. As we mentioned before, you can add a new tool to your selection through the menu bar. It'll appear in the tool list and you can then enable the tool by clicking the checkbox.

### Selection

#### Triangle selector by area

Selects all the triangles from the input that have an area smaller or greater than a value or an area that falls between two values. (illustration 6)

#### Triangle selector by edge length

Selects all the triangles from the input that have one or more edges with lengths smaller or greater than a value or lengths that fall between two values. (illustration 7)

#### Triangle selector by average pixel value

Selects triangles based on the average pixel value of all the pixels in the image that are represented by a triangle. This can then be compared to see if it's smaller or greater than a value or between two values. This tool only works when both a triangulation and an image are loaded. (illustration 8)

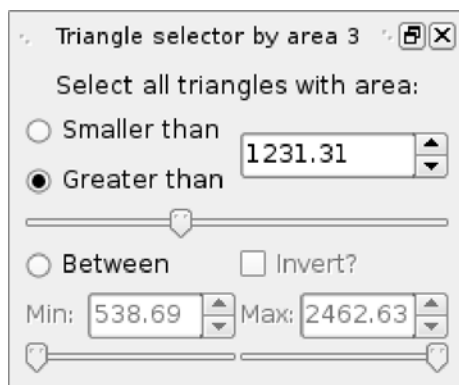


Illustration 6: Triangle selector by area

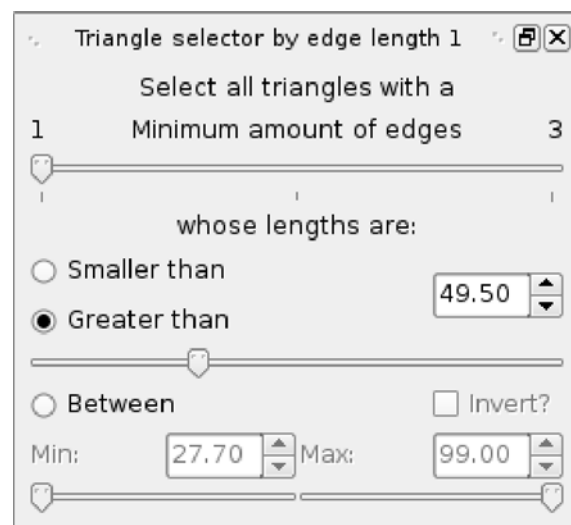
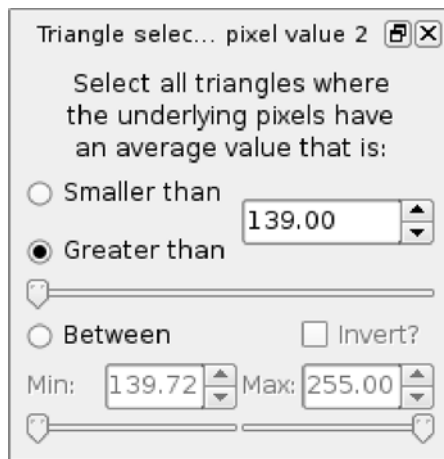
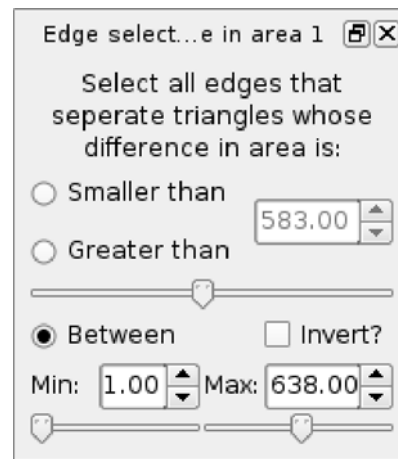


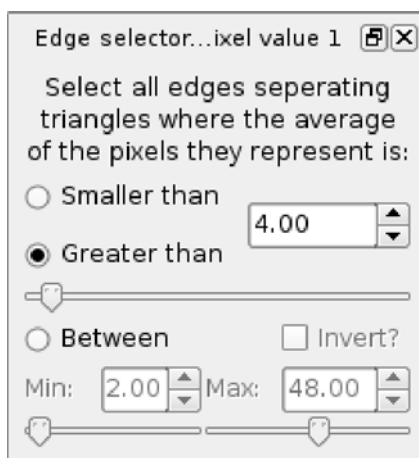
Illustration 7: Triangle selector by edge length



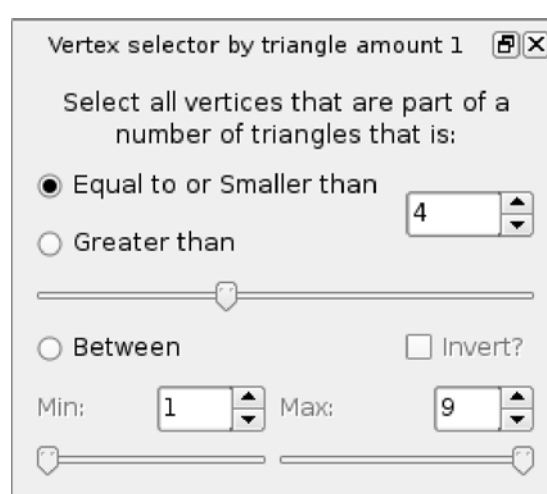
*Illustration 8: Triangle selector by average pixel value*



*Illustration 9: Edge selector by difference in area*



*Illustration 10: Edge selector by difference in average pixel value*



*Illustration 11: Vertex selector by triangle amount*

### Edge selector by difference in area

Enables you to select edges, based on the difference in area of the two triangles it separates. The difference can, again, be smaller or greater than a value or fall between two other values. (illustration 9)

### Edge selector by difference in average pixel value

Performs the same as 'Edge selector by difference in area', only here it functions by determining the difference in average pixel value. (illustration 10)

### Vertex selector by triangle amount

Selects all vertices that are part of an amount of triangles smaller than and equal to a value or greater than that value or if that amount falls between two values. (illustration 11)

### Triangle remover

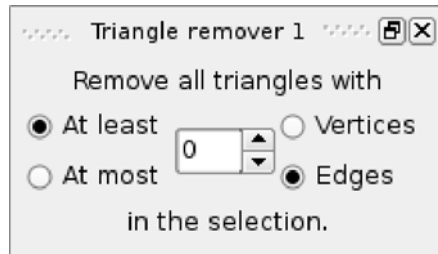
Removes all triangles from the selection that contain a specified amount of vertices or edges within the current selection. (illustration 12)

## Edge remover

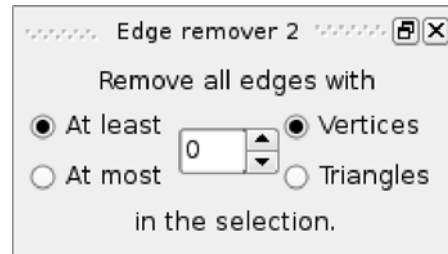
Performs the same as the 'Triangle remover', but on edges. (illustration 13)

## Vertex remover

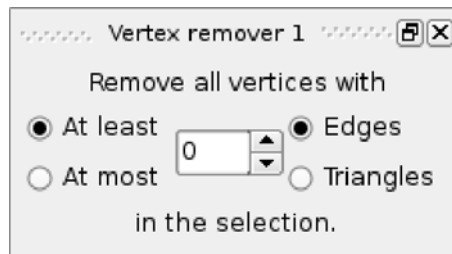
Performs the same as the 'Triangle remover', but on vertices. (illustration 14)



*Illustration 12: Triangle remover*



*Illustration 13: Edge remover*



*Illustration 14: Vertex remover*

## Edges from triangle boundary

Selects all the edges that separate a triangle in the current selection and a triangle not in the current selection, creating a boundary made out of edges of the current selection. This tool is automatic and therefore has no configurable settings.

## Internal vertices from triangles

Selects all the vertices that are only part of triangles that are in the selection. This tool is automatic and therefore has no configurable settings.

## Output

### Color by average pixel value

Colors all the triangles in the current selection according to the average value of all the pixels that are represented by the triangle. This tool is automatic and therefore has no configurable settings.

### Selection colorizer

Colors the vertices, edges and triangles in the selection according to the settings. For each, you can choose a color (by clicking the colored button) and opacity (by sliding the ruler). For edges, you can also choose how thick they are displayed and you can choose one of six styles in which to display the vertices. (illustration 15)

## Histogram of triangles by area

Creates a histogram of all the triangles, sorted by their area. Either the amount of bins or the size of the bins can be specified. (illustration 16)

## Histogram-based triangle colorizer by area

Functions like 'Histogram of triangles by area', only has the added function of coloring all the triangles belonging to a bin in the histogram. Just click on the bin to select a new color to draw all the triangles with. (illustration 17)

## Histogram of triangles by average pixel value

Functions like 'Histogram of triangles by area', only for the average pixel value of the pixels represented by a triangle. This tool only works when both a triangulation and an image have been loaded into triAna. (illustration 18)

## Histogram-based triangle colorizer by average pixel value

Functions like 'Histogram-based triangle colorizer by area', only for the average pixel value of the pixels represented by a triangle. This tool only works when both a triangulation and an image have been loaded into triAna. (illustration 19)

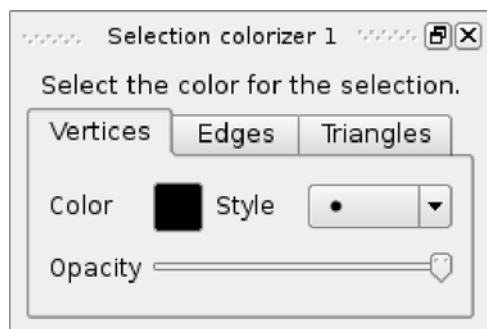


Illustration 15: Selection colorizer

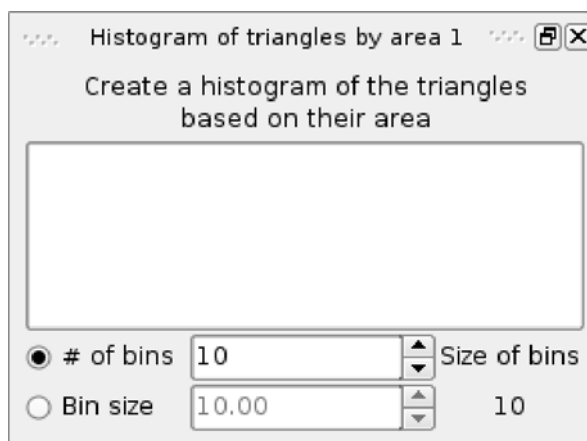


Illustration 16: Histogram of triangles by area

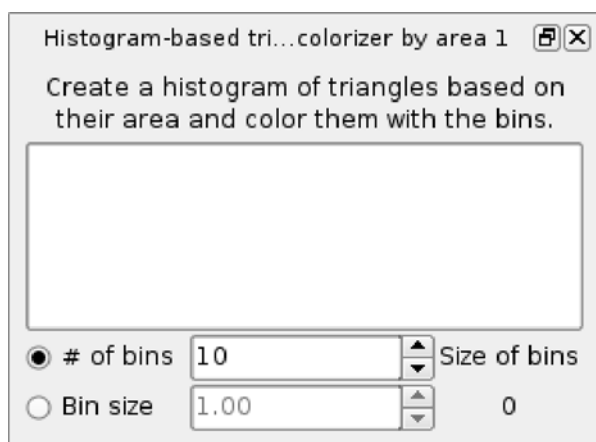


Illustration 17: Histogram-based triangle colorizer by area

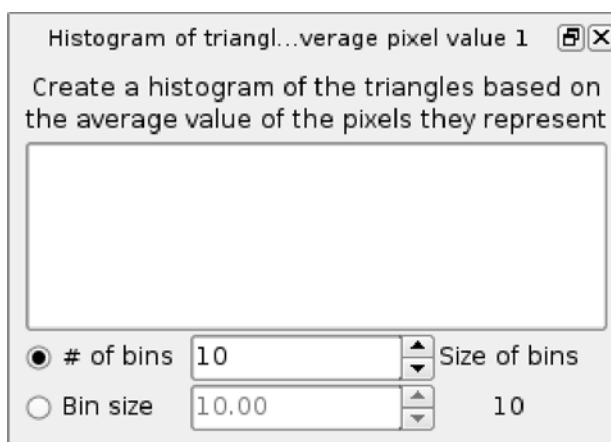
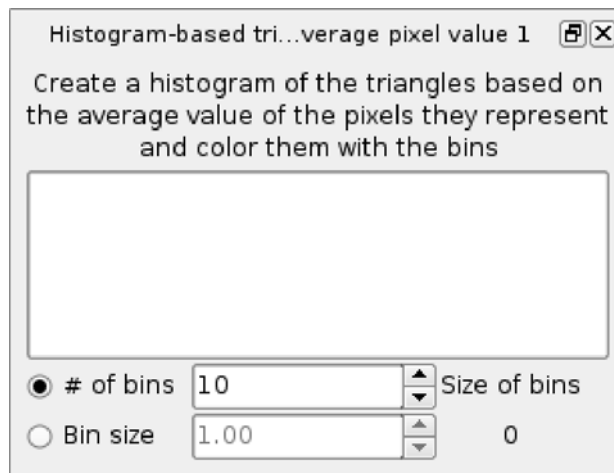


Illustration 18: Histogram of triangles by average pixel value



*Illustration 19: Histogram-based triangle colorizer by average pixel value*

## Example

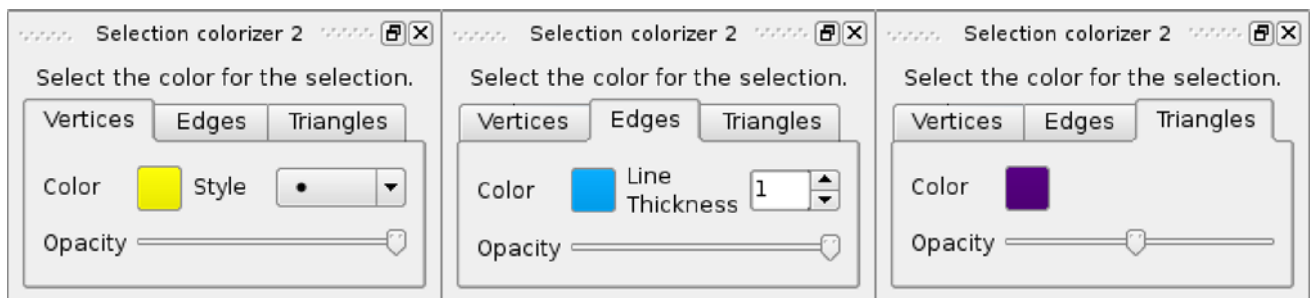
As an example of how triAna works, we'll try to select the triangles that represent the tree rings of an image of a tree stump. The rings are usually represented by the smaller triangles, so we'll work under that assumption.

### Step 1

First we start triAna and load the image and the triangulation file. Note the very detailed triangulation. Calculations on triangulations of this magnitude take a long time. To speed up the analysis, we'll disable 'Live updates' for now and press 'Update' after each step.

### Step 2

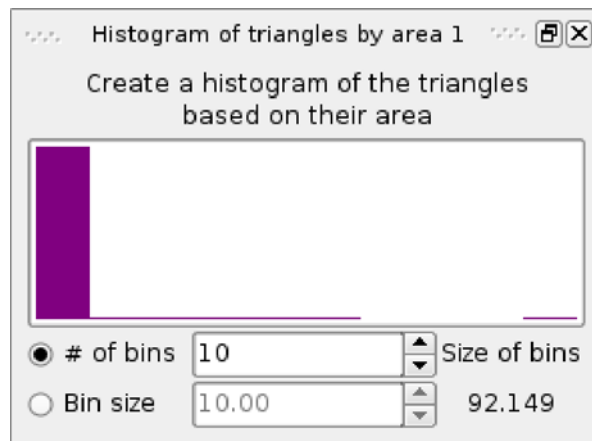
Since we want to check our progress at times, let's select the 'Selection colorizer' and enable it with extra setting for the edges and vertices (illustration 20). We have color! (Lots of it, since the initial selection is the entire triangulation.)



*Illustration 20: Color settings*

### Step 3

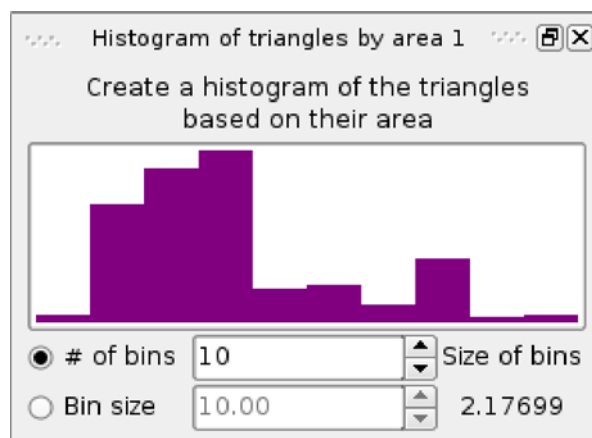
Because we know very little about the current triangulation, let's select 'Histogram of triangles by area' and enable it (illustration 21). Since most of the triangles are contained in the first bin our next step will be to select all the triangles that are contained in that bin.



*Illustration 21: An histogram of the current settings*

#### Step 4

After selecting 'Triangles by area' from the 'Tools'-menu and enabling it, we change the settings to 'Smaller than 93' (the min and max of the first bin) and check the histogram. Still, most of the triangles are contained in the first couple of bins, so we'll change our selection to all triangles 'Smaller than 24' (illustration 22).

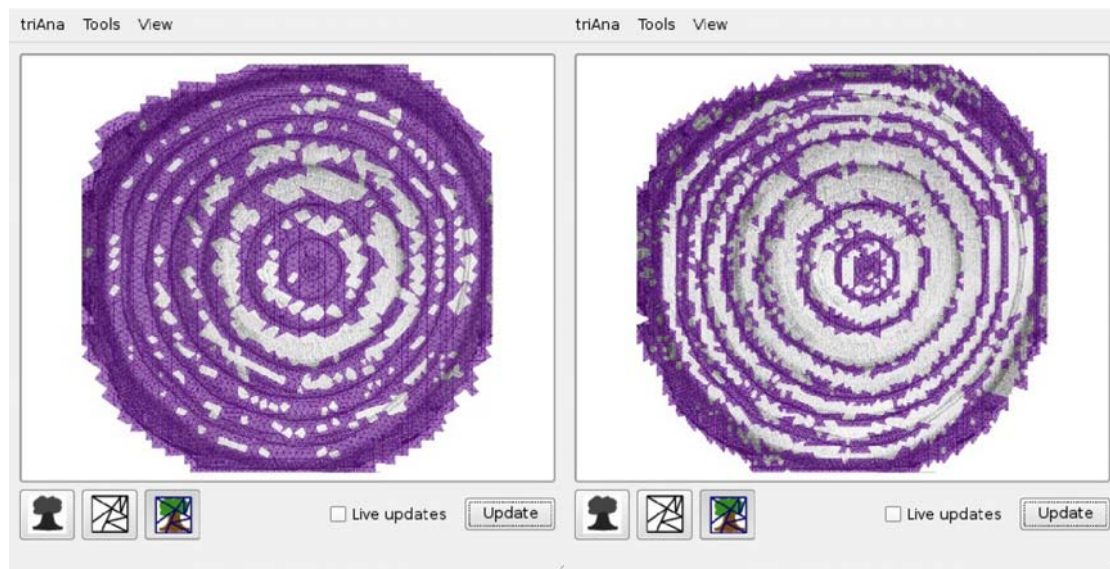


*Illustration 22: Updated histogram*

#### Step 5

Setting the opacity of the vertices and edges to 0 (they're cluttering up the image) we see that this is still too much. We set our filter size to 'Smaller than 12' and see that this improves the situation a lot (illustration 23).

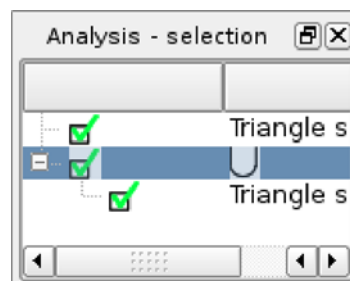




*Illustration 23: From triangles with area smaller than 24 to smaller than 12*

### *Step 6*

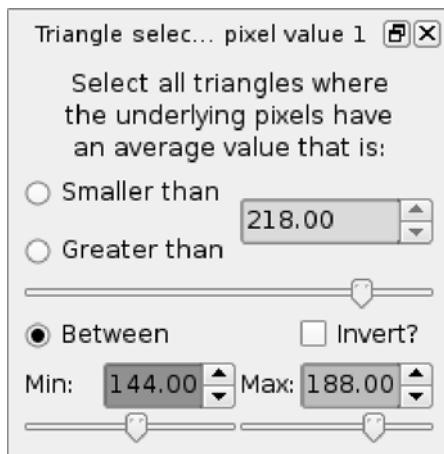
Though the selection has improved, it still doesn't adequately select the tree rings. There's a lot of bark selected and quite a lot of triangles that have nothing to do with the rings whatsoever. We know that the ring and the bark are darker than the rest of the tree and the bark is even a bit darker than the rings. We'll take advantage of that fact. Let's insert an 'Intersection' group after our 'Triangles by area' selection. We put the 'Triangles by average pixel value'-tool in this group (illustration 24).



*Illustration 24: The Intersection group with the tool*

### *Step 7*

After some experimentation, we set the settings of 'Triangles by average pixel value' to 'Between 144.00 and 188.00' (illustration 25). After enabling the group and the tool, this leaves us with a much better result (illustration 26).



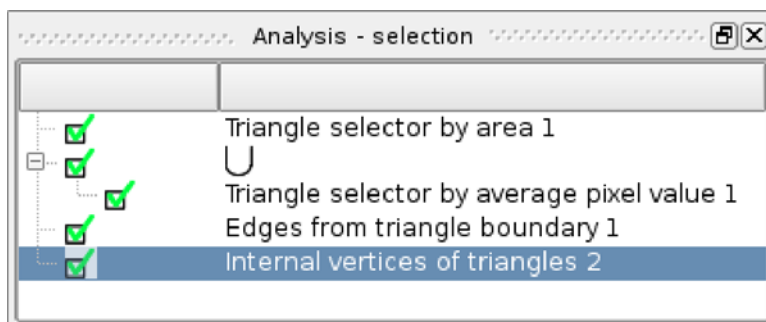
*Illustration 25: Settings of 'Triangles by average pixel value'*



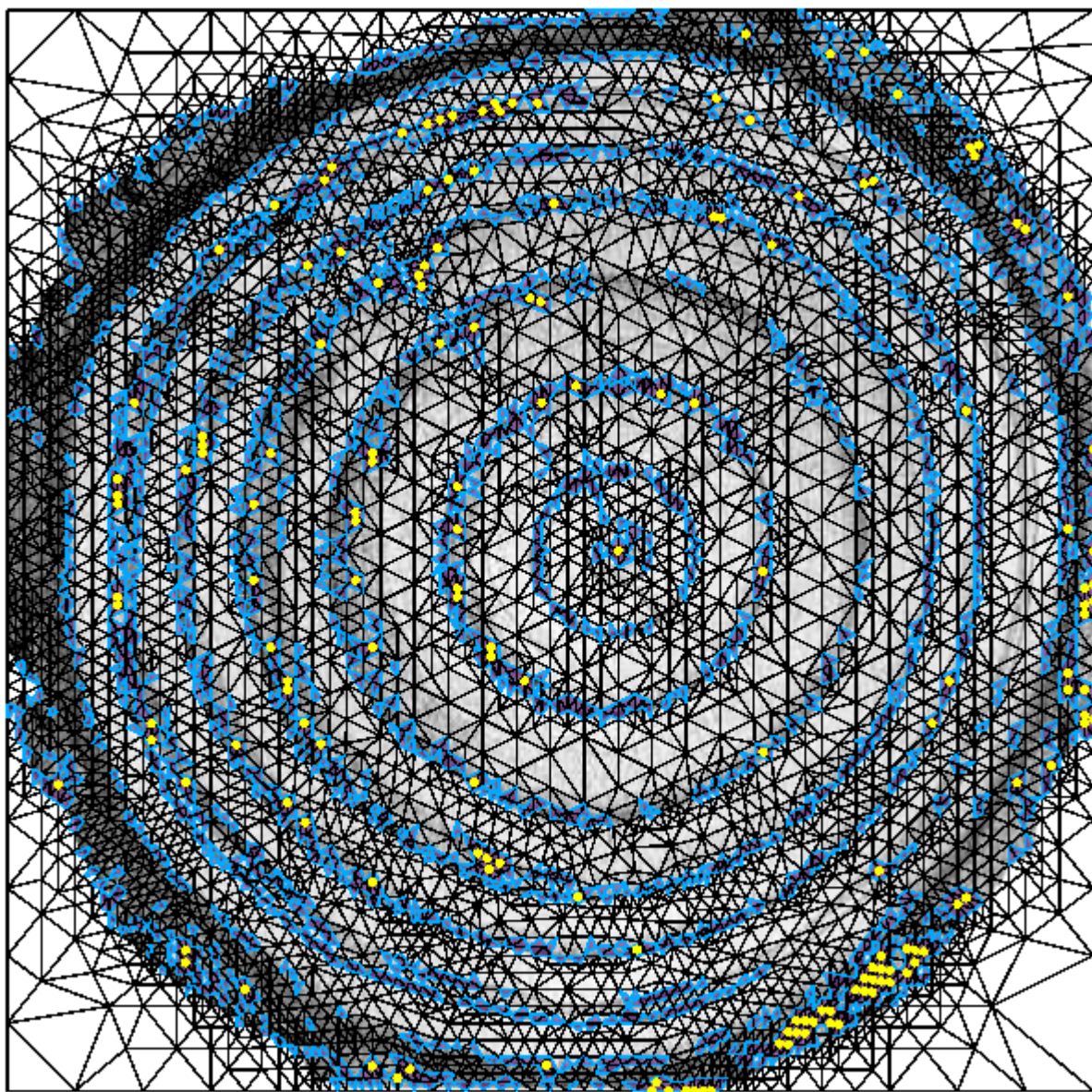
*Illustration 26: A much better selection of tree rings*

### Step 8

As the final step, we add the automatic tools 'Edge boundary of triangles' and 'Internal vertices of triangles' to our selection and enable these (illustration 27). After setting the opacity of both vertices and edges to maximum again, we get our end result: a rather adequate selection of the tree rings, with their borders and internal points (illustration 28).



*Illustration 27: The sequence of all the tools in this example*



*Illustration 28: The final result*

**Appendix B: Developer's Manual**



# triAna

## An image triangulation visualization and analysis tool

### version 1.0

### Developer's Manual

A bachelor project in cooperation with  
University of Chile  
Delft University of Technology

by  
Thomas Schaap (#1150561)  
Bruno Scheele (#1150588)

Supervisors  
N. Hitschfeld Kahler (University of Chile)  
P.R. van Nieuwenhuizen (Delft University of Technology)  
M. Sepers (Delft University of Technology)

## Table of Contents

- [Table of Contents](#)
- [Overview](#)
- [Plugins](#)
- [TriAnaPlugin](#)
  - [Example](#)
  - [Compiling a plugin](#)
  - [Steps in creating a plugin](#)
- [ImageInputPlugin](#)
  - [Example](#)
  - [Purpose](#)
  - [const QStringList mimetypes\( \) const](#)
  - [bool canLoad\( QIODevice &device \)](#)
  - [QImage loadImage\( QIODevice &device \)](#)
  - [QString mimetype\(\) const](#)
- [TriangulationInputPlugin](#)
- [AnalysisPlugin](#)
  - [Example](#)
  - [Properties](#)
  - [const QStringList informationEntries\( \) const and friends](#)
  - [void setInterface\( TriAnaView\\* interface, QWidget\\* parent \)](#)
  - [void setAnalysis\( Analysis\\* analysis \)](#)
  - [AnalysisPluginSetProperty setInteractiveProperty\( const QString& name, QVariant value \)](#)
  - [signal void informationChanged\(\)](#)
  - [signal void propertyChanged\(\)](#)
  - [void removeWidget\(\)](#)

- [TriAnaView\\* m\\_interface](#)
  - [Analysis\\* m\\_analysis](#)
  - [QMutex m\\_mutex](#)
- [CriteriaPlugin](#)
  - [Example](#)
  - [QList< TriangulationPart\\* > createSelection\( QList< TriangulationPart\\* > selection \)](#)
  - [SelectionGroup\\* parentGroup\( \) const](#)
- [OutputPlugin](#)
  - [Example](#)
  - [void setSelection\( QList< TriangulationPart\\* >& selection \)](#)
  - [void createOutput\( \)](#)
  - [void clearSelection\( \)](#)
  - [QList< TriangulationPart\\* >& m\\_selection](#)
- [Plugins with actions](#)
  - [Example](#)
  - [Using actions](#)
- [Plugins with widgets](#)
  - [Example](#)
  - [Using widgets](#)
  - [Caveats](#)
- [Frontends](#)
  - [Batch processing: the savefile runner](#)
  - [The basics: the example in detail](#)
  - [Beyond the basics: complex interfaces](#)

## Overview

triAna is a powerful program for researchers to work with. It would not be nearly as powerful, however, if it wasn't as extensible as it is. All of triAna is set up to allow additions using external parts, which creates a flexible program that can analyze the way you want, visualize the way you want and even look the way you want.

Extending triAna is not a very hard thing to do. Many things have been already been done and care is taken to keep the plugin framework easy to understand and use. All information needed to do so can be found in the API documentation and using this manual learning that API should not prove difficult.

Creating a frontend for triAna might pose a little challenge, depending on what the frontend should be able to do. A simple frontend that opens a savefile and uses it to process a triangulation/image pair won't be very difficult, but a full-fledged analysis tool as the standard frontend provides will cost quite some time. Using this manual should speed up learning the backend, in order to minimize the time needed to build such a frontend.

In order to read this documentation, it is recommended to first run doxygen in both the root of the triAna source directory and the directory of this file. These will generate the documentation of the backend, ui, plugins and widgets and the documentation of the examples.

## Plugins

The plugins that are used in triAna are all created as Qt plugins, implementing a couple of interfaces. To make the API consistent, triAna uses two interfaces that provide common functionality: TriAnaPlugin for all plugins and AnalysisPlugin for all plugins that perform selections or output. The four plugin types that derive from these parent interfaces are ImageInputPlugin, for loading images, TriangulationInputPlugin, for loading triangulations, CriteriaPlugin, for creating

selections, and OutputPlugin, for generating output.

## TriAnaPlugin

### Example

All plugins need to implement TriAnaPlugin. It's a simple interface giving a consistent way of looking at plugins that work with TriAna. Most likely learning to use TriAnaPlugin is most easy by example, so let's build one of the plugins that's delivered with triAna: the QImageReaderInput, which will allow us to open image types that are supported by Qt.

- [MyUselessImageReaderInput.h](#)
- [MyUselessImageReaderInput.cpp](#)

These two files will already give you a correct plugin, though it won't do much yet. You can, however, compile and load it. In a moment we will discuss the details of these files.

### Compiling a plugin

A CMakeLists.txt is built to search for Qt and search for triAna, setting all variables needed to create the plugin. For more information on CMake files, refer to CMake's documentation. It can be found in the examples directory.

Compiling the above will result in a library holding the plugin. This plugin, though it does nothing, will already load when pointed to in triAna.

### Steps in creating a plugin

The steps needed to create the plugin are rather easy and follow the standard Qt Plugin system. To make things easy, here's a checklist:

- Include the header of the interface you wish to inherit;
- Publicly inherit the interface;
- Put Q\_OBJECT in your class;
- Put Q\_INTERFACES in your class, with the interface you inherit and its superinterfaces;
- Declare your plugin using Q\_EXPORT\_PLUGIN2.

That's about all there is to it. Be sure to have a moc-file generated and included in the source and all should go well.

As far as implementing the interface is concerned, there's only a few things to be said. First of all, read the documentation of the different functions that need to be inherited well. They contain examples of correct and incorrect return values. Most important here is to keep in mind that the plugin should be usable by end-users in a real world setting. It's useful to know that the name of most plugins is actually shown to the user to select them from the menu 'Tools' in the standard UI. Also, the description is used there to show users what the plugin really does: it's the tooltip of every menu-item where it says so.

## ImageInputPlugin

### Example

Now that there's a working TriAnaPlugin, it would be nice to have it actually do something. For this purpose the MyImageReaderInput will inherit ImageInputPlugin to become a plugin that can read images. First of all the new files.

- [MyImageReaderInput.h](#)
- [MyImageReaderInput.cpp](#)

There haven't been many changes to the header-file. Note the new superclass of MyImageReaderInput and the extra interface in Q\_INTERFACES. Apart from that the methods of ImageInputPlugin have been added to the class, along with the private variable m\_mimetype.

When this plugin is compiled, a useful and working plugin is the result. It has actually already proven to be very useful: it's similar to the plugin that triAna uses by default to open images.

## Purpose

The purpose of an ImageInputPlugin is to provide triAna with the possibility to open image files of the formats it supports. When loaded, it will be queried automatically to open an image when needed.

## **const QStringList mimetypes( ) const**

The mimetypes method returns the mimetypes an ImageInputPlugin believes it will be able to open. It's not necessary to provide this, but it can speed up the loading of images.

## **bool canLoad( QIODevice &device )**

This function is always called before calling loadImage(). The ImageInputPlugin can read ahead in the device to see if it can actually read it. Be careful about what is returned here: false means that the device can't be read by the ImageInputPlugin and loadImage() won't be called. True means that the ImageInputPlugin can load the image in the device and loadImage() will be called subsequently. Make sure that loadImage() returns the loaded image or you might end up with erroneous behaviour.

## **QImage loadImage( QIODevice &device )**

The loadImage method is the heart of every ImageInputPlugin. When the call to canLoad() returns true, this function will be called to do the actual loading of the image. It's the ImageInputPlugin's task to read the image and to return it as a QImage.

## **QString mimetype() const**

Since it might be useful for users and frontends to know what type of image was opened, the ImageInputPlugin might be queried for that information after reading an image. This function should return the mimetype of the last successfully loaded image.

## TriangulationInputPlugin

TriangulationInputPlugin and ImageInputPlugin are very similar. Learning to use ImageInputPlugin is definitely enough to learn TriangulationInputPlugin. There is only one real difference: TriangulationInputPlugin's loadTriangulation returns a pointer to a Triangulation and will return 0 when loading failed, where



ImageInputPlugin returns QImage() when loading fails, since it returns a QImage object. Apart from this little difference, implementing a TriangulationInputPlugin is exactly the same as implementing an ImageInputPlugin.

One detail about TriangulationInputPlugin's does require mentioning: responsibility for consistency. When constructing a Triangulation from lists with Vertex, Edge and Triangle objects, the Triangulation will not check if they are consistent. Yet they really need to be. It's up to the TriangulationInputPlugin to see to this. See also Triangulation's documentation for more information.

## AnalysisPlugin

AnalysisPlugin is the superinterface for CriteriaPlugin and OutputPlugin. It defines the interface for all plugins that need to interact with the user, the frontend and the complete Analysis.

### Example

Using AnalysisPlugin directly is technically possible, but definitely not useful. Also, frontends could get confused with an advanced plugin that really does nothing. Therefore no direct example will be given. Instead AnalysisPlugin's methods are well visible in the examples for CriteriaPlugin and OutputPlugin.

### Properties

Implementors of AnalysisPlugin have to declare all their settings as properties to Qt as types that can be converted to a string or a stringlist. In doing so, an AnalysisPlugin creates immediate support for persistence and for having changes to properties that can be undone.

### **const QStringList informationEntries( ) const and friends**

Many plugins might have some things to tell the user. How many triangles they have selected, how small the smallest edge is or the statistical spread of all triangles divided by their own edges. Whatever it is they want to provide the user with, as long as it's data that can be put in a single string, information entries are most likely the way to go.

Information entries are calculated values that give some useful information to the user. They are provided by the plugin by means of informationEntries() and friends. informationEntries() will return a list with the names of all information entries that the plugin provides. Note that these names will most likely be shown to the user, so cryptic names are not useful. There are no restrictions to the name, either, just that long names are more likely not to integrate in some frontends well.

informationEntryDescription() is the function that provides a description about the information entries. This description is solely for the end user, who can read it in order to better understand the information entries contents.

The value of an information entry is provided by informationEntryValue(). This function returns the value of the information entry as a string. Nothing will be done with the string other than showing it to the user, so any formatting can be done without regard for converting back.

### **void setInterface( TriAnaView\* interface, QWidget\* parent )**

setInterface() is the function where an AnalysisPlugin is made aware of the frontend. For simple plugins a standard implementation is given that will simply remember the interface and to do some administration to ensure proper termination. More advanced plugins will want to override this function to tell the interface about widgets or actions it wants to show the user. Such plugins are advised, though, first to call their superclass's implementation of this function. This will take some work from their hands and make things generally easier.

After calling the superclass's implementation of this function, an AnalysisPlugin can provide the interface with widgets or actions as it sees fit. Be sure, though, to check if an interface was really provided: this might be 0. For all possibilities, see the documentation of TriAnaView's functions.

Whenever this function is called, plugins should assume earlier calls to have become invalid. In other words: if an AnalysisPlugin learns about frontend X via this functions and later on learns about frontend Y via another call to this function, it must assume X to have been destroyed, along with everything it gave to X.

The function interface() queries the plugin for the last interface that was set. If a plugin calls its superclasses's setInterface(), this function has been taken care of.

Note that when this function is called, the interface must already have loaded any image and triangulation data it wants the plugin to work with, in the case the plugin is being loaded from a savestate. In other words: you don't need to worry about using the triangulation and image provided by the interface when you expect values from the savestate to be given to you: the image and triangulation that need to be loaded from that savestate should be loaded and ready before this function is called.

## **void setAnalysis( Analysis\* analysis )**

Anticipating highly advanced plugins, an AnalysisPlugin is also told about the Analysis object it is part of. Currently, this is only used as a shortcut for quickly retrieving the Analysis an AnalysisPlugin is part of via the function analysis(), since no such highly advanced plugins have been thought of. Unless you're absolutely sure that you need this, you can easily rely on the standard implementation of this function.

## **AnalysisPlugin SetProperty setInteractiveProperty( const QString& name, QVariant value )**

setInteractiveProperty() is a very useful and important function. For others, mostly. This function uses the properties that have been declared to Qt to change settings of the plugin in a way that implements the Command pattern. The second argument to this function is optional, to allow code that calls this function to first create the AnalysisPlugin SetProperty for changing the value, only to provide it with a value later on.

When all settings of the plugin have been declared as Qt properties, like they should, the standard implementation takes care of everything.

## **signal void informationChanged( )**

An AnalysisPlugin that has information entries should keep the ones looking at those entries informed about changes. This is done via the signal/slot mechanism of Qt. Simply emitting this signal will tell all who wish to know that information entries have changed. This should be done whenever one of those values changes.

## **signal void propertyChanged( )**

An AnalysisPlugin will most likely have several Qt properties declared with its properties. Whenever one of these changes, this signal should be emitted. This will signal all those who want to know that settings of the plugin have changed. Emit this signal whenever the value of one the properties changes. Aggregate calls, for example one call at the end of a function that changes twenty properties, are OK.

## **void removeWidget( )**

This function is used for cleaning up whenever the plugin is destroyed or the interface it is registered with is destroyed. If the plugin first calls its superclass's setInterface() from its own setInterface() or doesn't have one of its own, this function is called when the interface is destroyed. Plugins who have widgets are not only encouraged to implement this function, but also to call it from their constructors.

Implementing this function is only useful for those plugins that have widgets they provide to an interface. In this function those widgets should be recalled. Multithreading is an issue here, which leads to the advice to use m\_mutex to synchronize.

Boilerplate code, as used by the widgets triAna gives by default, is this:

```
if( m_mutex.tryLock( ) && m_interface && m_widget ) {  
    m_interface->removeWidget( m_widget );  
    m_widget->deleteLater( );  
    m_widget = 0;  
    m_mutex.unlock( );  
}
```

The plugins using this boilerplate code have exactly one widget, stored in m\_widget. The rest of the variables used here are provided by AnalysisPlugin.

Using removeWidget( ) is also usefull when you provide the interface with actions. A single call to remove those widgets, safeguarded like the calls in the boilerplate, will ensure that no acture are left behind when the plugin is destroyed or the interface is destroyed.

## **TriAnaView\* m\_interface**

After calling the superclass's setInterface(), plugins can find the current interface in this variable. Be sure to check if it's 0.

## **Analysis\* m\_analysis**

In the rare case it might be needed, this variable holds the Analysis object the plugin belongs to after the standard implementation of setAnalysis() has been called. No guarantee is given as to when that is, though. Be sure to check if it's 0.

## **QMutex m\_mutex**

In order to synchronize access to the above variables, plugins can use this recursive mutex. Although it would be safest to surround all calls to these variables with calls to the mutex, it is most important in the destructor and the function removeWidget( ), since race-conditions are guaranteed to exist there.

## **CriteriaPlugin**

A CriteriaPlugin is a plugin that wishes to influence the selection made in an Analysis. In an Analysis, first a selection is created by creating a set of all parts of the triangulations (triangles, edges and vertices) and then CriteriaPlugins are allowed to create subsets of that subset.

Creating a CriteriaPlugin consists of nothing more than creating an AnalysisPlugin with one extra function.

## Example

Let's start out with an example again. It's a CriteriaPlugin that wishes to select only the biggest triangle and leaves all edges and vertices in, as is recommended.

- [BigTriangleSelector.h](#)
- [BigTriangleSelector.cpp](#)

When looking through the source, you'll notice the absolute absence of any reference to AnalysisPlugin. This is because AnalysisPlugin has default implementations for all its functions, to ease the stress on developers of simple plugins like this one. Indeed, this plugin has only one more method than a bare TriAnaPlugin.

### **QList< TriangulationPart\* > createSelection( QList< TriangulationPart\* > selection )**

This function is the core of a CriteriaPlugin: a selection goes in, a subselection goes out. Plugins creating selections must be consistent in how they do so. In other words, they must take care to produce the same result when presented with the same selection and under the same circumstances (loaded image, settings, and others like that). It is therefore absolutely forbidden to create a CriteriaPlugin that decides if a part of the selection is accepted by testing (someRandomGenerator() == 42). Although the effects might be funny in an interactive interface, it's not very useful.

A part that will be needed rather often is the image behind a part of the triangulation. To get to that, look at the interface in `m_interface`, especially methods `TriAnaView::image( )` and `TriAnaView::pixels( )` are often useful.

Note that a plugin is obliged to create a *subselection* of the provided selection. Although this requirement is not technically necessary, it is there nonetheless to ensure that thinking about selection plugins remains easy. Besides that, there are other ways available for growing the selection, such as using a union.

### **SelectionGroup\* parentGroup( ) const**

CriteriaPlugins can be grouped together using a SelectionGroup. The hierarchy is maintained by Analysis and the SelectionGroup makes sure the set-operations are applied, but the CriteriaPlugin should know, and provide, in which group it belongs. The `parentGroup( )` function returns this group, or 0 if there is no such group. The group is set using `setParentGroup( )`.

No reasons are known for plugins to need to override this function. The standard implementation simply records the parent in the variable `m_parent`.

## OutputPlugin

An OutputPlugin is a plugin that wishes to create output, a visualization, or otherwise do something dependent on the final selection created in an Analysis. About the only thing an OutputPlugin may not do is change that selection.

## Example

The simple example of an OutputPlugin gives a visualization of the selected triangles by making them all red.

- [RedTriangles.h](#)
- [RedTriangles.cpp](#)

This example is very simple. It's once again visible that AnalysisPlugin takes care of everything for us, leaving us to just concentrate on the important part: creating red triangles.

### **void setSelection( QList< TriangulationPart\* >& selection )**

This function is called when the CriteriaPlugins have done creating the selection. The only thing that happens here is setting the selection. No output may be generated in this function. It is guaranteed this function will be called at least once before createOutput( ) is called.

The standard implementation of this function places the selection in m\_selection. This will most likely be enough for most plugins.

### **void createOutput( )**

The function createOutput( ) defines the core of most OutputPlugins. It is here that the OutputPlugin creates its output, such as changing properties of the geometry, calculating and showing statistics or anything else that can be repeated lots of times, based on the selection that was set during the last call to setSelection( ). Using this selection will be a matter of using the variable m\_selection for most OutputPlugins.

This function is the only place where an OutputPlugin is allowed to change the properties of the geometry objects, i.e. all vertices, edges and triangles in the triangulation.

### **void clearSelection( )**

This function is used as an easy way to clear the set selection, for example when the triangulation it is based on is destroyed. Subclasses need normally not bother with this function, but when they wish to react to this event, implementing this function is a good way of achieving that.

When, for some reason, an OutputPlugin might need to clear the selection of its own, calling this function is the best way to do so.

### **QList< TriangulationPart\* >& m\_selection**

The selection as last set by the function setSelection( ). This variable is always valid in the function createOutput( ).

## Plugins with actions

Plugins might want to do things that can't be done many times in a row. Such actions include printing to a physical printer, writing a file and doing extremely expensive calculations. Basically anything one would normally only want to do once or maybe twice counts as such an action.

The most usual plugin having this kind of requirement is an `OutputPlugin`. However, using the standard function `createOutput()` is not really the best place for actions such as these. It may be called many times. The standard UI for triAna, for example, normally calls it for every update that has been made in the `Analysis` object, including all its plugins. Although some may think having a hardcopy printed of every step along the way in their research, most will think it's a bad idea. Therefore, `AnalysisPlugins` have the ability to provide `QAction` objects to the interface.

## Example

A real-world example usage of this is the creation of a triangulation based on a loaded image using an external tool. We don't want to do this with every update, so we supply the interface with an action called 'Generate triangulation' that we will listen to. The user can then choose when to generate the triangulation.

- [TriangulationGenerator.h](#)
- [TriangulationGenerator.cpp](#)

When compiling and loading this plugin, you can add it to an `Analysis` and, when you've loaded an image, you can use the action it provided to generate a triangulation from that image. In the standard UI this is done by clicking right on the plugin and choosing "Generate triangulation" from the menu.

Since no actual program is easily available to do such triangulations, the plugin will not perform the triangulation itself, but apologize to the user and ask the user to do the triangulation using the generated image file.

## Using actions

Using actions is very easy. Just create `QActions` as you would normally do and give them to the interface in the function `setInterface()`. Be sure to remove them when you're being destroyed or when the interface is destroyed. To this end you can use the function `removeWidget()`.

## Plugins with widgets

Many plugins will want to provide some UI to the user, for example to show graphs or to allow the user to conveniently configure the plugin. This is enabled by allowing `AnalysisPlugin` subclasses to provide the interface with `TriAnaWidgets`. A `TriAnaWidget` is a normal `QWidget` that has an `AnalysisPlugin` as owner.

Providing widgets is a common and powerful way of interacting with the end-user directly. Do be careful, however, to avoid the caveats these widgets bring with them.

## Example

We extended the earlier `RedTriangles` to color all triangles in a configurable color. The configuration can be done via the `TriAnaWidget` it provides.

- [ColoredTriangles.h](#)
- [ColoredTriangles.cpp](#)
- [ColoredTrianglesWidget.h](#)
- [ColoredTrianglesWidget.cpp](#)

These are the source files for both the plugin and the widget the plugin uses. The layout of the widget itself, although it's very simple, was built using Qt Designer and is saved in file called `widget.ui`. The CMake file is slightly extended to compile the

widget and include the widget class. Both files can be found in the directory of the example.

When you compile and load this plugin in triAna, you can use it as an output for your analysis. When you click on the output, you will see the widget pop up in a dockwidget.

## Using widgets

Using widgets is really simple: just build the widget and tell your plugin to create an instance of it when the function `setInterface()` is called. Then provide that instance to the interface and, if the interface can handle it, it will be used. Also make sure to remove the widget when needed. Put a call to `removeWidget()` in your destructor and implement that function accordingly to take care of this.

There are no real rules about how to write your widget, only that they should be able to return their owner. This helps an interface to identify which plugin provided a widget. Also, very large widgets might pose a problem to some users.

Please note that quickly adding and removing widgets is not a good idea: add your widgets once during the call to `setInterface()`, that's by far the most secure. If there's a need to add at different moments, though, be sure to make it thread-safe using the mutex provided by `AnalysisPlugin` and don't expect your widget to pop up directly. In the standard UI a provided widget won't be shown until the user clicks on the plugin in the `Analysis`.

## Caveats

A couple of caveats exist when using widgets. To prevent implementors to easily fall into them, here's a list of them with descriptions.

- **Default values;** Don't set default values on your widget. Although in many cases your widget will be instantiated together with a new instance of your plugin, it can as well be instantiated while loading a save file. In the latter case the values in your plugin will be set before the widget is instantiated and the defaults in your widget will give a wrong view.
- **Listen to the plugin;** Don't assume the plugin will be configured using only the widget. It can very well be configured using other methods, such as `setInteractiveProperty()`. Have the widget listen to the signals of the plugins to update your widget when the plugin changes.
- **Test sizing policies well;** There are several ways to have Qt calculate the size of something and not all are the best way. Experiment with policies to see which fits your widget best. The size policy that's used for the standard triAna plugin widgets is both horizontally and vertically fixed.
- **Looping signals;** Watch out for signals being caught to emit other signals that emit the previous signal again. Several ways exist to create this nasty kind of loop. A surefire way is having your widget listen to the `propertyChanged()` signal of your plugin and having it set a value in your plugin whenever one of its controls is used. If no checking is done if the value is the same as had already been set earlier, you'll find the widget updating the plugin, which updates the widget, which updates the plugin, etc.
- **Signals from updates have side-effects;** It might be very useful to use, for example, both a slider and spinbox for the same value, to allow your users to quickly navigate big values, but also to allow accurate changes or direct entering of exact values. The same loop that is described above will still lead to trouble here, even when checking for the same values. Consider the slider and the spinbox. The slider contains integer values and the spinbox floating point values. The actual value that's used is the floating point

value. Now when changing the spinbox from 0.3 to 5.5, you suddenly see the value become 5.0.

What happened? The moment you changed the spinbox it changed the value in the plugin, which emitted the `propertyChanged()` signal. This signal was connected to a slot that sets the new values of both the spinbox and the slider. The spinbox is no problem, but the moment the slider was set, the value was first rounded down to 5, since it needs to be integer. With the slider having its value changed to 5 it emitted a signal which was picked up as if it was changed by a user and which was processed accordingly: the new value of the slider was set in the plugin. Normally this would cause nothing if the plugin checks if the value is the same as that which was already set, but now it first saw 5.5 and later on 5, which is different, so it changed once again.

There's many such side-effects you all need to take into account. The solutions are usually rather easy, but trying to figure out what happens can be difficult. Some known solutions are using Qt's `disconnect()` and `connect()` to temporarily stop listen to changes in the slider to fix the case above and using the `editingFinished()` signal instead of the `valueChanged()` signal of the spinboxes to allow them to autorepeat.

## Frontends

The functional part of triAna, the backend, is set up in a way to allow many different ways of interacting with users. There are many uses of the backend. It's set up to be quite fast, so batch processing is possible. Also, the full-fledged functionality of the standard UI might not always be needed and a simple viewer might speed things up a lot. Even a full text-mode UI for analysing triangulations is imaginable, although most likely very difficult to write.

Writing a frontend can range from a breeze to a lot of work. We'll show an example to get you started. Not all aspects of writing a frontend will be discussed here, since many of them do not concern the triAna API, but rather the Qt libraries. Also, not every class in the triAna backend will be discussed in all aspects, though with a complex frontend you will touch most of them.

### Batch processing: the savefile runner

Most likely the most easy to build frontend is one that simply opens a savefile and writes the output of it all to some file. This is exactly what the savefile runner does.

- [SavefileRunner.h](#)
- [SavefileRunner.cpp](#)

The output of this little utility is slightly naive: it builds an XML file with an element for every triangle, which has an element for every edge, which has an element for every vertex. The output is thus not really useful. More useful output, however, costs a lot of extra code and is not within the scope of the example.

The example is more or less the absolute minimum to load a savefile and use it. Many aspects, such as widgets, displaying and actions, are simply ignored by the example and any interaction with the user is absent.

### The basics: the example in detail

When we take a look at the header-file, there's a couple of things that need to be done for every frontend.

1. The main class that inherits `TriAnaView`, also inherits `QObject`, directly or



indirectly.

2. 6 public and 4 protected methods must be implemented. These define the public and internal API of every TriAnaView.
3. Some boilerplate code with some signals, slots and methods is needed let TriAnaView function as it should. The reason for this boilerplate code is that TriAnaView is no QObject of its own, since it's desirable to have subclasses of TriAnaView inherit other QObjects as well, such as a QMainWindow. TriAnaView still needs to function as a QObject, though, and the boilerplate makes this possible.

The boilerplate code in the headerfile is this:

```
signals:
    void triangulationChanged( );
    void imageChanged( );
    void closing( );
protected slots:
    virtual void cleanupCache( ) { TriAnaView::cleanupCache( ); }
public:
    QObject* asQObject( ) { return this; }
```

Apart from these items and the boilerplate code, the header file contains what any QObject header file would have: it's own methods, a Q\_OBJECT declaration, its variables and whatever else you see fit.

The source-code of the example offers some more insight in how to use the backend. We'll highlight the important points again.

1. A main method with a QApplication. Even in non-gui versions, QApplication must be used to allow plugins to build their widgets.
2. In the constructor setup should be done as usual. An added point to think about is preloading the plugins in the standard plugin directory. PluginLoader gives all the necessary functions to do so (loadablePlugins() and loadPlugin()). For most applications this is a good idea.
3. The destructor that emits closing. Any plugin may provide widgets to the interface, but these need to be removed as well, when the time comes. Due to internal matters plugins can't reliably do so without ruining either themselves or the interface. This signal fixes that problem. Simply emit it from your destructor and all should be well.
4. The call to loadAnalyses() in runSavefile(). This call shows how easy loading and saving savefiles is. Simply call loadAnalyses() and much will be done without the need to write code.
5. The calls to the Analysis objects in runSavefile(). These calls show how Analysis encapsulates the use of plugins. Once set up, all one needs to do is call the slots run() and doOutput() to have the selection created and output done.
6. The functions triangulation() and image(). These functions return the current triangulation and image. These functions are meant for the backend to access the triangulation and image easily.
7. The functions addWidget() and removeWidget(), which are not implemented here, provide the interface with widgets from plugins. Plugins call these functions to provide and remove widgets. Be sure to keep track of them, if you accept them.
8. The functions addAction() and removeActions(), which are not implemented here, provide the interface with actions from plugins. Plugins call these functions to provide and remove actions. Be sure to keep track of them, if you accept them.
9. The functions loadImage() and loadTriangulation() are called by the backend during loading to tell the interface an image or triangulation should be loaded. The interface can decide for itself if and how to load the image or triangulation. When implementing more complex frontends, you'll want to load the image or triangulation and keep track of it, until finalizeLoadAnalyses() is called before

actually using it anywhere.

10. The function `addAnalysis()`. This function may be called many times or not at all, all depending on the savefile that is being loaded. Here, also, the frontend may decide what to do with the provided Analyses. When implementing more complex frontends, you'll want to keep track of them, until `finalizeLoadAnalyses()` is called before actually using them anywhere.
11. The function `finalizeLoadAnalyses()`. This function is called after any needed calls to `addAnalysis()`, `loadImage()` and `loadTriangulation()` and only if the loading was succesfull. In complex frontends, this is the place to start using the Analyses, image and triangulation that were provided through those 3 methods.

## Beyond the basics: complex interfaces

There are many things to take into consideration when writing a frontend. Lots of them have been mentioned above and the rest can be found in the documentation of the backend. A list of questions will help building complex frontends. These questions won't be discussed here, since the answer to them is very application-specific and all that needs to be done for a particular answer can be found in the documentation of the backend.

1. Can the user interact with an Analysis?  
And if so:
  1. Which functions can the user use?
  2. Will the user be allowed to use more than one Analysis at the same time?
  3. Will the user be able to use groups in the selection using `SelectionGroup`?
2. How will the user interact with plugins? Will you use the provided widgets and/or actions?
3. What output will you create?
4. When will you run an analysis and when will you generate output?
5. Can the user influence which image and/or triangulation is being used?
6. What plugins are available and can the user influence this?

When creating a frontend, especially a complex one, it's important to keep in mind that you don't control the order of events. They might be different another time. This means you can't really depend on the order in which things happen, apart from when documented. A good example of this is when a plugin is added and when it provides you with its widgets: the order of these two actions depends on why the plugin was created.

## Appendix C: Examples' documentation

Contains the following source codes:

- ◆ BigTriangleSelector.h
- ◆ BigTriangleSelector.cpp
- ◆ ColoredTriangles.h
- ◆ ColoredTriangles.cpp
- ◆ ColoredTrianglesWidget.h
- ◆ ColoredTrianglesWidget.cpp
- ◆ MyImageReaderInput.h
- ◆ MyImageReaderInput.cpp
- ◆ MyImageReaderInput.h
- ◆ MyImageReaderInput.cpp
- ◆ MyUselessImageReaderInput.h
- ◆ MyUselessImageReaderInput.cpp
- ◆ RedTriangles.h
- ◆ RedTriangles.cpp
- ◆ SavefileRunner.h
- ◆ SavefileRunner.cpp
- ◆ TriangulationGenerator.h
- ◆ TriangulationGenerator.cpp

```
#ifndef BIGTRIANGLESELECTOR_H
#define BIGTRIANGLESELECTOR_H

#include <QtPlugin>
#include <CriteriaPlugin.h>
#include <QList>

class TriangulationPart;

class BigTriangleSelector : public CriteriaPlugin {
    Q_OBJECT
    Q_INTERFACES( TriAnaPlugin AnalysisPlugin CriteriaPlugin )
public:
    QList< TriangulationPart* > createSelection( QList< TriangulationPart* > selection );

    const QString name( ) const;
    const QString description( ) const;
    const QString id( ) const;
    TriAnaPlugin* newInstance( QObject* parent = 0 );
};

#endif
```

```

#include "BigTriangleSelector.h"

#include <TriangulationPart.h>
#include <Triangle.h>

QList< TriangulationPart* > BigTriangleSelector::createSelection( QList< TriangulationPart* >
selection ) {
    QList< TriangulationPart* > res;
    // Go over all parts
    double maxArea = 0;
    Triangle* maxTriangle = 0;
    for( QList< TriangulationPart* >::Iterator it = selection.begin( ); it != selection.end( );
it++ ) {
        // If this is a triangle, we'll need to compare it to the biggest we've seen so far
        Triangle* t = qobject_cast< Triangle* >( *it );
        if( t ) {
            if( t->area( ) > maxArea ) {
                maxArea = t->area( );
                maxTriangle = t;
            }
        }
        // If it's not a triangle, we won't touch it
        else {
            res.append( *it );
        }
    }
    // Let's see if we found a triangle at all. If we did, the biggest is in maxTriangle and we
    should add that.
    if( maxTriangle )
        res.append( maxTriangle );
    // Done! Let's return the result.
    return res;
}

const QString BigTriangleSelector::name( ) const {
    return "Big Triangle Selector";
}

const QString BigTriangleSelector::description( ) const {
    return "Selects the biggest triangles out of all selected triangles.";
}

const QString BigTriangleSelector::id( ) const {
    return "org.bar.foo.BigTriangleSelector";
}

TriAnaPlugin* BigTriangleSelector::newInstance( QObject* parent ) {
    BigTriangleSelector* instance = new BigTriangleSelector( );
    if( parent )
        instance->setParent( parent );
    return instance;
}

Q_EXPORT_PLUGIN2( bigtriangleselection, BigTriangleSelector )

#include "BigTriangleSelector.moc"

```

```
#ifndef COLOREDTRIANGLES_H
#define COLOREDTRIANGLES_H

#include <QtPlugin>
#include <OutputPlugin.h>
#include <QColor>

class TriAnaView;
class ColoredTrianglesWidget;

class ColoredTriangles : public OutputPlugin {
    Q_OBJECT
    Q_INTERFACES( TriAnaPlugin AnalysisPlugin OutputPlugin )
    Q_PROPERTY( QColor color READ color WRITE setColor )
public:
    ColoredTriangles( );
    ~ColoredTriangles( );

    void createOutput( );

    void setInterface( TriAnaView* interface, QWidget* ui_parent );

    const QString name( ) const;
    const QString description( ) const;
    const QString id( ) const;
    TriAnaPlugin* newInstance( QObject* parent = 0 );

    QColor color( ) const;
    void setColor( QColor color );

protected:
    void removeWidget( );

private:
    ColoredTrianglesWidget* m_widget;
    QColor m_color;
};

#endif
```

```

#include "ColoredTriangles.h"
#include <TriangulationPart.h>
#include <Triangle.h>
#include <QList>
#include "ColoredTrianglesWidget.h"
#include <TriAnaView.h>

ColoredTriangles::ColoredTriangles( ) : m_widget( 0 ), m_color( QColor( 0, 0, 255 ) ) {
}

ColoredTriangles::~ColoredTriangles( ) {
    removeWidget( );
}

void ColoredTriangles::setInterface( TriAnaView* interface, QWidget* ui_parent ) {
    OutputPlugin::setInterface( interface, ui_parent );
    if( m_interface ) {
        m_widget = new ColoredTrianglesWidget( this, ui_parent );
        m_interface->addWidget( m_widget );
    }
}

void ColoredTriangles::removeWidget( ) {
    if( m_mutex.tryLock( ) && m_interface && m_widget ) {
        m_interface->removeWidget( m_widget );
        m_widget->deleteLater( );
        m_widget = 0;
        m_mutex.unlock( );
    }
}

QColor ColoredTriangles::color( ) const {
    return m_color;
}

void ColoredTriangles::setColor( QColor color ) {
    if( color == m_color )
        return;
    m_color = color;
    // A property of ours has changed: tell the world!
    // Note that we don't specifically mention the widget here, even though it should be updated.
    // We rely on the widget, instead, to listen to propertyChanged( ).
    emit propertyChanged( );
}

void ColoredTriangles::createOutput( ) {
    // Go over all parts of the selection
    for( QList< TriangulationPart* >::Iterator it = m_selection.begin( ); it != m_selection.end( ); it++ ) {
        // If this is a Triangle, we should really fill it with our color
        Triangle* t = qobject_cast< Triangle* >( *it );
        if( t ) {
            t->setColor( m_color );
            t->setFilled( true );
        }
    }
}

const QString ColoredTriangles::name( ) const {
    return "Colored Triangles";
}

const QString ColoredTriangles::description( ) const {
    return "Colors all triangles in a configurable color";
}

const QString ColoredTriangles::id( ) const {
    return "org.bar.foo.ColoredTriangles";
}

TriAnaPlugin* ColoredTriangles::newInstance( QObject* parent ) {

```

```
    ColoredTriangles* instance = new ColoredTriangles( );  
    if( parent )  
        instance->setParent( instance );  
    return instance;  
}  
  
Q_EXPORT_PLUGIN2( coloredtriangles, ColoredTriangles )  
  
#include "ColoredTriangles.moc"
```



```
#ifndef COLOREDTRIANGLESWIDGET_H
#define COLOREDTRIANGLESWIDGET_H

// This file will be generated by Qt's uic, called via CMake:
#include "ui_widget.h"

#include "ColoredTriangles.h"
#include <TriAnaWidget.h>
#include <QColor>

class AnalysisPlugin;

class ColoredTrianglesWidget : public TriAnaWidget, public Ui_widget {
    Q_OBJECT
public:
    ColoredTrianglesWidget( ColoredTriangles* parent, QWidget* ui_parent = 0 );

    AnalysisPlugin* owner( );

private slots:
    void buttonColorClicked( );
    void refreshInterface( );

private:
    ColoredTriangles* m_parent;
};

#endif
```

```
#include "ColoredTrianglesWidget.h"
#include <AnalysisPlugin.h>
#include <QPalette>
#include <QColorDialog>

ColoredTrianglesWidget::ColoredTrianglesWidget( ColoredTriangles* parent, QWidget* ui_parent ) :
TriAnaWidget( ui_parent ), m_parent( parent ) {
    setupUi( this );
    refreshInterface( );

    // Connect the button and listen to when the properties, i.e. color, of our parent has changed
    connect( buttonColor, SIGNAL( clicked( ) ), this, SLOT( buttonColorClicked( ) ) );
    connect( m_parent, SIGNAL( propertyChanged( ) ), this, SLOT( refreshInterface( ) ) );
}

AnalysisPlugin* ColoredTrianglesWidget::owner( ) {
    return m_parent;
}

void ColoredTrianglesWidget::buttonColorClicked( ) {
    // Get ourselves a new color
    QColor newColor = QColorDialog::getColor( m_parent->color( ), this );
    if( !newColor.isValid( ) )
        return;
    newColor = QColor( newColor.red( ), newColor.green( ), newColor.blue( ) );
    // Tell our parent about the new color
    m_parent->setColor( newColor );
    // Note the absence of any changes to ourselves: we'll do that in refreshInterface,
    // which will be run as a result of setting the color of our parent (which emit
    propertyChanged( ) )
}

void ColoredTrianglesWidget::refreshInterface( ) {
    // Color the button the same as the color set in our parent
    QPalette p = buttonColor->palette( );
    p.setColor( QPalette::Button, m_parent->color( ) );
    buttonColor->setPalette( p );
}

#include "ColoredTrianglesWidget.moc"
```

```
#ifndef MYIMAGEREADERINPUT_H
#define MYIMAGEREADERINPUT_H

#include <QtPlugin>
#include <ImageInputPlugin.h>

class QIODevice;
class QImage;

class MyImageReaderInput : public ImageInputPlugin {
    Q_OBJECT
    Q_INTERFACES( TriAnaPlugin ImageInputPlugin )
public:
    QImage loadImage( QIODevice& device );
    const QStringList extensions( ) const;
    const QStringList mimetypes( ) const;
    bool canLoad( QIODevice& device );
    QString mimetype( ) const;

    const QString name( ) const;
    const QString description( ) const;
    const QString id( ) const;
    TriAnaPlugin* newInstance( QObject* parent = 0 );

private:
    QString m_mimetype;
};

#endif
```

```

#include "MyImageReaderInput.h"

#include <QImage>
#include <QImageReader>

QImage MyImageReaderInput::loadImage( QIODevice& device ) {
    // Put the image ready to be read
    QImageReader img( &device );

    // Remember what format it was
    QString format = QString( img.format( ) ).toLowerCase( );
    if( format == "png" ) {
        m_mimetype = "image/png";
    }
    else if( format == "jpg" || format == "jpeg" ) {
        m_mimetype = "image/jpeg";
    }
    else if( format == "gif" ) {
        m_mimetype = "image/gif";
    }
    else {
        m_mimetype = QString( );
    }

    // Read the image
    QImage image = img.read();

    // Only remember the mimetype if loading was succesful
    if( image.isNull( ) )
        m_mimetype = QString( );

    // Return the loaded image
    return image;
}

bool MyImageReaderInput::canLoad( QIODevice& device ) {
    QImageReader img( &device );
    return img.canRead( );
}

const QStringList MyImageReaderInput::mimetypes( ) const {
    // Qt is not so nice as to give us the mimetypes themselves, so we'll have to translate them
    QList< QByteArray > list = QImageReader::supportedImageFormats( );
    QStringList res;
    QList< QByteArray >::iterator it;
    QString ext;
    for( it = list.begin( ); it != list.end( ); it++ ) {
        ext = QString( *it ).toUpperCase( );
        if( ext == "GIF" ) {
            res.append( "image/gif" );
        }
        else if( ext == "JPG" ) {
            if( !res.contains( "image/jpeg" ) )
                res.append( "image/jpeg" );
        }
        else if( ext == "JPEG" ) {
            if( !res.contains( "image/jpeg" ) )
                res.append( "image/jpeg" );
        }
        else if( ext == "PNG" ) {
            res.append( "image/png" );
        }
    }
    // Return the mimetypes we should be able to read
    return res;
}

const QStringList MyImageReaderInput::extensions( ) const {
    QList< QByteArray > list = QImageReader::supportedImageFormats( );
    QStringList res;
    QList< QByteArray >::iterator it;

```

```
    for( it = list.begin( ); it != list.end( ); it++ ) {
        res.append( QString( *it ) );
    }
    return res;
}

QString MyImageReaderInput::mimetype( ) const {
    // Return the mimetype of the last succesfully loaded image
    return m_mimetype;
}

const QString MyImageReaderInput::name( ) const {
    return "Qt ImageReader";
}

const QString MyImageReaderInput::description( ) const {
    return "Reads images supported by Qt.";
}

const QString MyImageReaderInput::id( ) const {
    return "org.bar.foo.MyImageReaderInput";
}

TriAnaPlugin* MyImageReaderInput::newInstance( QObject* parent ) {
    MyImageReaderInput* instance = new MyImageReaderInput( );
    if( parent )
        instance->setParent( parent );
    return instance;
}

Q_EXPORT_PLUGIN2( myimagereaderinput, MyImageReaderInput )

#include "MyImageReaderInput.moc"
```

```
#ifndef MYUSELESSIMAGEREADERINPUT_H
#define MYUSELESSIMAGEREADERINPUT_H

#include <QtPlugin>
#include <TriAnaPlugin.h>

class MyUselessImageReaderInput : public TriAnaPlugin {
    Q_OBJECT
    Q_INTERFACES( TriAnaPlugin )
public:
    const QString name( ) const;
    const QString description( ) const;
    const QString id( ) const;
    TriAnaPlugin* newInstance( QObject* parent = 0 );
};

#endif
```

```
#include "MyUselessImageReaderInput.h"

const QString MyUselessImageReaderInput::name( ) const {
    return "Qt ImageReader";
}

const QString MyUselessImageReaderInput::description( ) const {
    return "Reads images supported by Qt.";
}

const QString MyUselessImageReaderInput::id( ) const {
    return "org.bar.foo.MyUselessImageReaderInput";
}

TriAnaPlugin* MyUselessImageReaderInput::newInstance( QObject* parent ) {
    MyUselessImageReaderInput* instance = new MyUselessImageReaderInput( );
    if( parent )
        instance->setParent( parent );
    return instance;
}

Q_EXPORT_PLUGIN2( myuselessimagereaderinput, MyUselessImageReaderInput )

#include "MyUselessImageReaderInput.moc"
```

```
#ifndef REDTRIANGLES_H
#define REDTRIANGLES_H

#include <QtPlugin>
#include <OutputPlugin.h>

class RedTriangles : public OutputPlugin {
    Q_OBJECT
    Q_INTERFACES( TriAnaPlugin AnalysisPlugin OutputPlugin )
public:
    void createOutput( );

    const QString name( ) const;
    const QString description( ) const;
    const QString id( ) const;
    TriAnaPlugin* newInstance( QObject* parent = 0 );
};

#endif
```



```
#include "RedTriangles.h"
#include <TriangulationPart.h>
#include <Triangle.h>
#include <QList>

void RedTriangles::createOutput( ) {
    // Go over all parts of the selection
    for( QList< TriangulationPart* >::Iterator it = m_selection.begin( ); it != m_selection.end(
); it++ ) {
        // If this is a Triangle, we should really fill it with red
        Triangle* t = qobject_cast< Triangle* >( *it );
        if( t ) {
            t->setColor( QColor( 255, 0, 0 ) );
            t->setFilled( true );
        }
    }
}

const QString RedTriangles::name( ) const {
    return "Red Triangles";
}

const QString RedTriangles::description( ) const {
    return "Colors all triangles red";
}

const QString RedTriangles::id( ) const {
    return "org.bar.foo.RedTriangles";
}

TriAnaPlugin* RedTriangles::newInstance( QObject* parent ) {
    RedTriangles* instance = new RedTriangles( );
    if( parent )
        instance->setParent( instance );
    return instance;
}

Q_EXPORT_PLUGIN2( redtriangles, RedTriangles )

#include "RedTriangles.moc"
```

```

#ifndef SAVEFILERUNNER_H
#define SAVEFILERUNNER_H

#include <QObject>
#include <QList>
#include <QImage>

#include <TriAnaView.h>
#include <Analysis.h>
#include <Triangulation.h>

class Triangulation;
class QImage;
class TriAnaWidget;

class SavefileRunner : public QObject, public TriAnaView {
    Q_OBJECT
public:
    SavefileRunner( );
    ~SavefileRunner( );
    /**
     * Stores the file data needed to do the loading, running and saving in runSavefile( ).
     * This function creates a timer with a timeout of 0 to call runSaveFile( ).
     *
     * @arg filename    The file to load.
     * @arg output      The file to write the result to.
     */
    void setFileData( const QString& filename, const QString& output );
private slots:
    /**
     * Loads the savefile, runs the Analyses it found on the image and triangulation it found and
     writes the result to the outputfile.
     */
    void runSavefile( );

// Here follow the methods every TriAnaView must implement
public:
    Triangulation* triangulation( );
    QImage& image( );
    bool addWidget( TriAnaWidget* widget );
    bool removeWidget( TriAnaWidget* widget );
    void addAction( QAction* action, AnalysisPlugin* owner );
    void removeActions( AnalysisPlugin* owner );
protected:
    bool addAnalysis( Analysis* analysis );
    bool loadImage( QIODevice& image, const QString& mimetype, const QString& filename, bool
embedded );
    bool loadTriangulation( QIODevice& triangulation, const QString& mimetype, const QString&
filename, bool embedded );
    void finalizeLoadAnalyses( );

// Here follow some private variables so we can remember all we're loading
private:
    Triangulation* m_triangulation;
    QImage m_image;
    QList< Analysis* > m_analyses;
    QString m_filename;
    QString m_output;

// Here follows some biolerplate code for each TriAnaView
// This code is necessary to make sure TriAnaView can operate as a QObject, while it technically
can't inherit from QObject
signals:
    void triangulationChanged( );
    void imageChanged( );
    void closing( );
protected slots:
    virtual void cleanupCache( ) { TriAnaView::cleanupCache( ); }
public:
    QObject* asQObject( ) { return this; }
};

```

```
#endif
```

```

#include "SavefileRunner.h"

#include <QFileInfo>
#include <QTextStream>
#include <QTimer>
#include <QImage>
#include <QDomDocument>
#include <QDomElement>
#include <QListIterator>
#include <QApplication>

#include <Triangulation.h>
#include <Triangle.h>
#include <Edge.h>
#include <Vertex.h>
#include <PluginLoader.h>
#include <ImageLoader.h>
#include <TriangulationLoader.h>

// A simple main method to start a QApplication and load our class
//
// We expect the last two arguments to be the savefile to load and the output-file to write,
// respectively.
int main( int argc, char** argv ) {
    // Although we won't provide a GUI ourselves, we still need to load a QApplication: the
    // plugins require the GUI part to be functional
    QApplication app( argc, argv );
    SavefileRunner runner;
    // Read the arguments and check if those are files.
    QString filename;
    QString output;
    bool bad = false;
    if( QCoreApplication::arguments( ).size( ) < 3 ) {
        bad = true;
    }
    else {
        filename = QCoreApplication::arguments( ).at( QCoreApplication::arguments( ).size( ) - 2 );
        output = QCoreApplication::arguments( ).last( );
        QFileInfo fil( filename );
        if( !( fil.exists( ) && fil.isFile( ) ) ) {
            bad = true;
        }
    }
    if( bad ) {
        // Tell the user how to use this program
        QTextStream err( stderr );
        err << "SavefileRunner example" << endl
            << "Usage: savefilerunner savefile outputfile" << endl;
        return -1;
    }
    // Tell the savefile runner about these files and get busy
    runner.setFileData( filename, output );
    return app.exec( );
}

SavefileRunner::SavefileRunner( ) : m_triangulation( 0 ) {
    // Let's load the plugins that were installed to the standard plugin directory
    QStringList plugins = PluginLoader::instance( )->loadablePlugins( );
    for( QStringList::Iterator it = plugins.begin( ); it != plugins.end( ); it++ ) {
        if( PluginLoader::instance( )->loadPlugin( *it ).isEmpty( ) ) {
            QTextStream err( stderr );
            err << "Warning: loading plugin " << *it << " failed:" << endl
                << PluginLoader::instance( )->errorString( ) << endl;
        }
    }
}

SavefileRunner::~SavefileRunner( ) {
    emit closing( );
}

```

```

void SavefileRunner::setFileData( const QString& filename, const QString& output ) {
    m_filename = filename;
    m_output = output;
    // Make sure my own slot is called as soon as the event-loop starts running
    QTimer::singleShot( 0, this, SLOT( runSavefile( ) ) );
}

void SavefileRunner::runSavefile( ) {
    QTextStream err( stderr );
    // Sanity check
    if( m_filename.isEmpty( ) || m_output.isEmpty( ) ) {
        // Since this is an internal error, just fail silently
        QApplication::exit( -1 );
        return;
    }

    // Load the file
    if( !loadAnalyses( m_filename ) ) {
        err << "Could not load savefile " << m_filename << ":" << endl
            << errorString( ) << endl;
        QApplication::exit( -1 );
        return;
    }

    // Check if we have a triangulation
    if( !m_triangulation ) {
        // Although really too bad, this is really a normal case, so return 0
        err << "No triangulation found in savefile " << m_filename << endl;
        QApplication::exit( 0 );
        return;
    }

    // Run all analyses
    for( QList< Analysis* >::Iterator it = m_analyses.begin( ); it != m_analyses.end( ); it++ ) {
        (*it)->run( );
        (*it)->doOutput( );
    }

    // Save the resulting triangulation to XML
    QFile output( m_output );
    if( !output.open( QIODevice::WriteOnly ) ) {
        err << "Could not write to output file " << m_output << ":" << endl
            << output.errorString( ) << endl;
        QApplication::exit( -1 );
        return;
    }

    QDomDocument document;
    QDomElement root = document.createElement( "triangulation" );
    QListIterator< Triangle* > triangles = m_triangulation->triangles( );
    while( triangles.hasNext( ) ) {
        Triangle* triangle = triangles.next( );
        QDomElement triangleElement = document.createElement( "triangle" );
        triangleElement.setAttribute( "color", triangle->color( ).name( ) );
        triangleElement.setAttribute( "filled", triangle->isFilled( ) );
        for( int edgeNumber = 0; edgeNumber < 3; edgeNumber++ ) {
            Edge* edge = (*triangle)[edgeNumber];
            QDomElement edgeElement = document.createElement( "edge" );
            edgeElement.setAttribute( "color", edge->color( ).name( ) );
            edgeElement.setAttribute( "visible", edge->isVisible( ) );
            edgeElement.setAttribute( "thickness", edge->thickness( ) );
            for( int vertexNumber = 0; vertexNumber < 2; vertexNumber++ ) {
                Vertex* vertex = (*edge)[vertexNumber];
                QDomElement vertexElement = document.createElement( "vertex" );
                vertexElement.setAttribute( "color", vertex->color( ).name( ) );
                vertexElement.setAttribute( "visible", vertex->isVisible( ) );
                vertexElement.setAttribute( "displaystyle", (int)vertex->displayStyle( ) );
                vertexElement.setAttribute( "x", vertex->x( ) );
                vertexElement.setAttribute( "y", vertex->y( ) );
                edgeElement.appendChild( vertexElement );
            }
        }
    }
}

```

```

        triangleElement.appendChild( edgeElement );
    }
    root.appendChild( triangleElement );
}
document.appendChild( root );

if( output.write( document.toByteArray( ) ) == -1 ) {
    err << "Could not write to output file " << m_output << ":" << endl
        << output.errorString( ) << endl;
    QCoreApplication::exit( -1 );
    return;
}

output.close( );

// We're done
QCoreApplication::exit( 0 );
}

Triangulation* SavefileRunner::triangulation( ) {
    return m_triangulation;
}

QImage& SavefileRunner::image( ) {
    return m_image;
}

bool SavefileRunner::addWidget( TriAnaWidget* /*widget*/ ) {
    // We ignore widgets
    return false;
}

bool SavefileRunner::removeWidget( TriAnaWidget* /*widget*/ ) {
    // We've already ignored widgets
    return false;
}

void SavefileRunner::addAction( QAction* /*action*/, AnalysisPlugin* /*owner*/ ) {
    // We are encouraged to give the user the opportunity to use an action
    // However, we're here for batch-work, without use intervention
    // So we choose to ignore actions
}

void SavefileRunner::removeActions( AnalysisPlugin* /*owner*/ ) {
}

bool SavefileRunner::addAnalysis( Analysis* analysis ) {
    m_analyses.append( analysis );
    return true;
}

bool SavefileRunner::loadImage( QIODevice& image, const QString& mimetype, const QString&
filename, bool embedded ) {
    QString extension;
    if( filename.indexOf( '.' ) != -1 ) {
        extension = filename.mid( filename.lastIndexOf( '.' ) + 1 );
    }
    m_image = ImageLoader::instance( )->loadImage( image, mimetype, extension );
    return true;
}

bool SavefileRunner::loadTriangulation( QIODevice& triangulation, const QString& mimetype, const
QString& filename, bool embedded ) {
    QString extension;
    if( filename.indexOf( '.' ) != -1 ) {
        extension = filename.mid( filename.lastIndexOf( '.' ) + 1 );
    }
    m_triangulation = TriangulationLoader::instance( )->loadTriangulation( triangulation,
mimetype, extension );
    return true;
}

```

```
void SavefileRunner::finalizeLoadAnalyses( ) {  
    // Since we don't have any data to replace, we did follow the advice not to change anything  
    // until this function is called  
    // So it can be left empty  
}  
  
#include "SavefileRunner.moc"
```

```
#ifndef TRIANGULATIONGENERATOR_H
#define TRIANGULATIONGENERATOR_H

#include <QtPlugin>
#include <OutputPlugin.h>

class TriAnaView;

class TriangulationGenerator : public OutputPlugin {
    Q_OBJECT
    Q_INTERFACES( TriAnaPlugin AnalysisPlugin OutputPlugin )
public:
    ~TriangulationGenerator( );

    void createOutput( );

    void setInterface( TriAnaView* interface, QWidget* ui_parent = 0 );

    const QString name( ) const;
    const QString description( ) const;
    const QString id( ) const;
    TriAnaPlugin* newInstance( QObject* parent = 0 );

protected:
    void removeWidget( );

private slots:
    void generateTriangulation( );
};

#endif
```



```

#include "TriangulationGenerator.h"
#include <TriAnaView.h>
#include <QImage>
#include <QDir>
#include <QTemporaryFile>
#include <QImageWriter>
#include <QFileDialog>
#include <QMessageBox>
#include <QAction>

TriangulationGenerator::~TriangulationGenerator( ) {
    // The right code is already in removeWidget, let's let that function to the job
    removeWidget( );
}

void TriangulationGenerator::createOutput( ) {
    // We don't do anything here, but we need to implement this function nonetheless
}

void TriangulationGenerator::setInterface( TriAnaView* interface, QWidget* ui_parent ) {
    OutputPlugin::setInterface( interface, ui_parent );
    // We wish to supply the interface with an action
    if( m_interface ) {
        QAction* action = new QAction( "Generate triangulation", this );
        connect( action, SIGNAL( triggered( ) ), this, SLOT( generateTriangulation( ) ) );
        m_interface->addAction( action, this );
    }
}

void TriangulationGenerator::removeWidget( ) {
    // Protect this function using the mutex m_mutex to prevent us from using a dangling pointer
    if( m_mutex.tryLock( ) && m_interface ) {
        m_interface->removeActions( this );
    }
}

void TriangulationGenerator::generateTriangulation( ) {
    // If we don't have an interface to ask the image from, don't bother
    if( !m_interface )
        return;

    // Get the image we need to use
    QImage image = m_interface->image( );
    // Without an image, no triangulation
    if( image.isNull( ) ) {
        QMessageBox::information( 0, "No image", "No triangulation was generated, since no image
was loaded. Please load an image and try again." );
        return;
    }

    // The external tool needs this as a file, save it first to a temporary file
    QTemporaryFile tempFile( QDir::temp().filePath( "triangulationGeneratorImageFileXXXXXX.png" )
);
    if( !tempFile.open( ) ) {
        QMessageBox::warning( 0, "Could not open tempfile", QString( "No triangulation was
generated, since the temporary file to save the image to could not be opened: %1" ).arg(
tempFile.errorString( ) ) );
        return;
    }

    QImageWriter writer( &tempFile, "PNG" );
    if( !writer.write( image ) ) {
        QMessageBox::warning( 0, "COuld not write tempfile", QString( "No triangulation was
generated, since the image could not be written to the temporary file: %1" ).arg(
writer.errorString( ) ) );
        return;
    }

    // Call the external program to generate the triangulation
    QString filename = QFileDialog::getSaveFileName( 0, "Generate triangulation", "." );
    if( filename.isEmpty( ) ) {

```

```
    // The user pressed cancel
    return;
}

// We don't have a program that actually does generate triangulations, so let's ask the user
to do so
QMessageBox::information( 0, "Generate triangulation", QString( "A program-call should be
inserted here to generate the triangulation. Alas, no such program-call is currently available and
all I can do is ask you to open image that is saved temporarily in %1 and to generate the
triangulation yourself, saving it in %2, before pressing the 'OK'-button below." ).arg(
tempFile.fileName( ) ).arg( filename ) );
}

const QString TriangulationGenerator::name( ) const {
    return "Triangulation generator";
}

const QString TriangulationGenerator::description( ) const {
    return "Uses an external tool to generate a triangulation from the currently loaded image.";
}

const QString TriangulationGenerator::id( ) const {
    return "org.bar.foo.TriangulationGenerator";
}

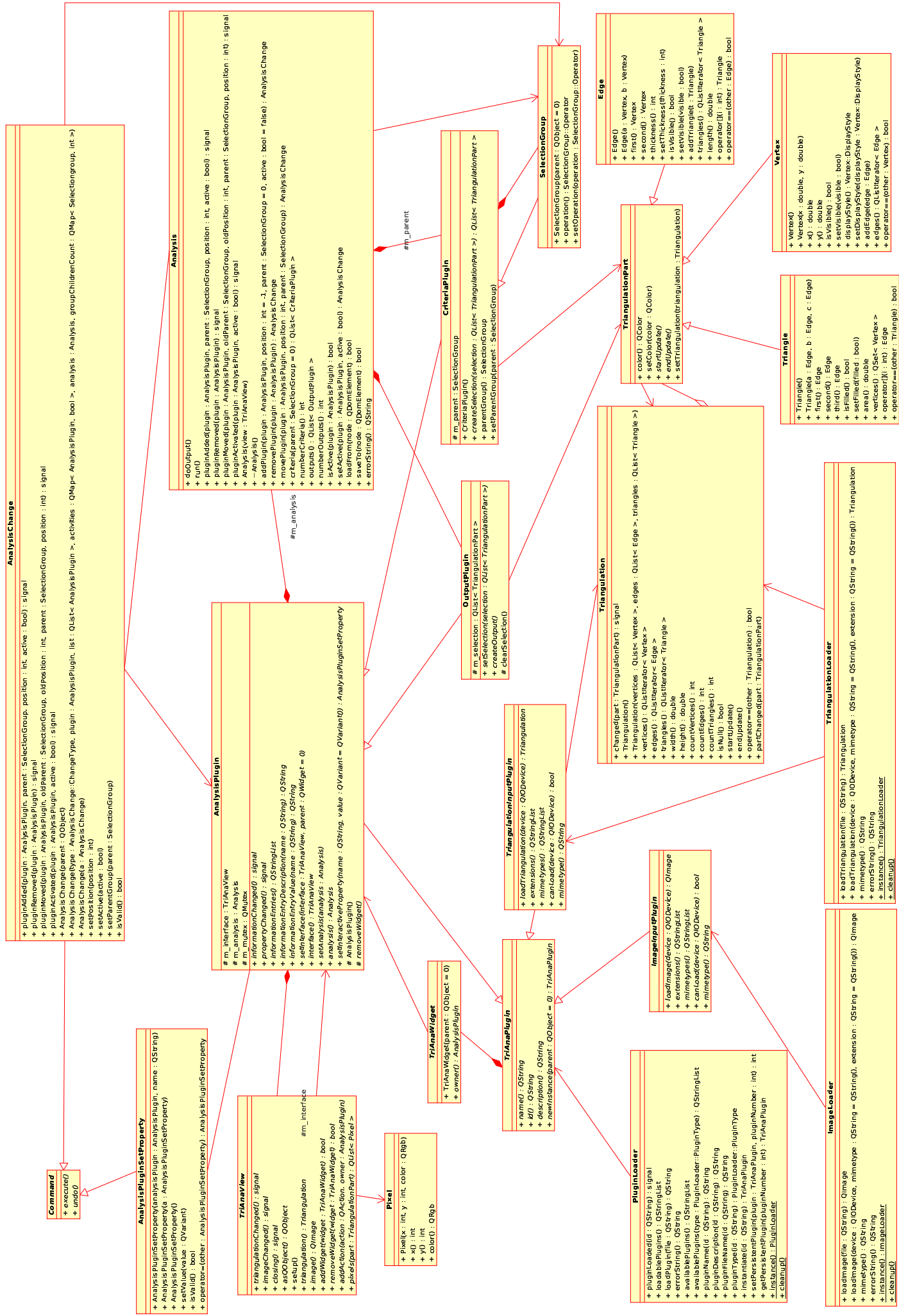
TriAnaPlugin* TriangulationGenerator::newInstance( QObject* parent ) {
    TriangulationGenerator* instance = new TriangulationGenerator( );
    if( parent )
        instance->setParent( parent );
    return instance;
}

Q_EXPORT_PLUGIN2( triangulationgenerator, TriangulationGenerator )

#include "TriangulationGenerator.moc"
```

## **Appendix D: The UML-model of triAna**

In the UML model several methods are marked as abstract. Several of these abstract methods do have default implementations. This difference in interpretation is due to the limitation in UML to differentiate between what are in C++ called virtual and pure virtual functions. The abstract functions in the UML model are the ones that can be easily overridden in subclasses (virtual methods).



**Appendix E: XML format for save files**

**triAna**

**Save files – format description**

**Version 1.0**

Last changed 05-07-07

## **1. Introduction and terminology**

In order to have a consistent and clean way of saving any work done in triAna, this description has been written as a reference to the organisation of savefiles for triAna. This document can be used to both read and write those documents, as long as one understands how XML works.

This document references several other documents, using them as a basis for describing the savefiles of triAna. All of them are referenced inline as well as in the index at the end of this document.

Whereever the words 'must', 'must not', 'required', 'shall', 'shall not', 'should', 'should not', 'recommended', 'may' and 'optional' occur, these are to be interpreted as described in RFC 2119, found at <http://www.rfc-editor.org/rfc/rfc2119.txt>.

## **2. Main document**

This section describes how a triAna savefile looks like without going into its content. Properties such as media, standards and the base are covered.

### **2.1 Physical description**

triAna savefiles must be bytestreams saved in files or other media. These bytestreams should be encoded in UTF8, described in RFC 3629 located at <http://tools.ietf.org/html/rfc3629>, but any encoding will do as long as it can contain the necessary data and is completely freely available to anyone who wishes to use it.

When saved in files, the extension for triAna savefiles should be '.trs'. This extension is an abbreviation of **tri**Ana savefile and has been chosen over the more logical alternative '.tri' to avoid conflicts with triangulation files, which happen to have the logical extension '.tri'.

### **2.2 General format**

The general format of a triAna savefile is an XML-document. The XML-standard is available on <http://www.w3.org/TR/xml/> and is required for correct interpretation of information in this document.

#### **2.2.1 Namespace**

To avoid conflicts with other XML documents, a separate namespace has been declared for use with triAna. This namespace is called 'triAna'. Namespaces for XML are described at <http://www.w3.org/TR/REC-xml-names/> and are recommended for correct interpretation of triAna savefiles.

The namespace URI of triAna is '<http://triAna.dcc.uchile.cl/>'.

When a triAna document is saved, prefixing the element-names with the namespace is recommended, whether the namespace is actually processed or not.

When a triAna document is loaded, it is recommended not to rely on the namespace prefix actually being there. Implementations are recommended to disregard any part up to and including the first colon in the name of an element if such a colon appears in the name and namespaces are not processed.

All elements and attributes in this specification fall within the triAna namespace and must therefore be prefixed with 'triAna:' whenever used in a savefile. For clarity in this specification, this prefix is not mentioned elsewhere.

#### **2.2.2 Root element**

The name of the root element of a triAna savefile is 'savefile'. Such an element may also be used as a child of another element, to embed the triAna savefile within a document. It is recommended for implementations to also provide a way to load and save from such an embedded source, but this is



not required and still the interpretation of the rest of the source is completely up to the implementation.

### **3. savefile element**

This section describes the contents of a savefile element. The savefile element is the root element of a triAna savefile and hence contains all needed information.

A savefile element should contain no information other than the children that are described here. If other information is included, this information must be ignored by implementations.

Allowed children are:

- last-image-source;
- last-triangulation-source;
- analysis.

All analysis-elements together will be referred to as the analysis-set. The analysis element is described in chapter 4.

A savefile element has exactly one attribute, named version. If any other attributes occur, implementations must ignore these.

#### **3.1 last-image-source**

This element describes the source of the image that has been used with the analysis-set. Implementations may use this information to provide a quick way for users to continue working on the same image with this analysis-set. The last-image-source element is not required and must not occur more than once in a savefile element.

The last-image-source element comes in two forms. Implementations are recommended to implement both, but only the referring form is required. When the embedded form is not implemented, it must still be detected and disregarded when found while loading a savefile.

If a last-image-source is found, but cannot be loaded due to restrictions such as access, availability, recognition or any other reason, the last-image-source element must be ignored.

##### **3.1.1 referring last-image-source**

This form provides a reference to a source where the image is located. It has one required attribute named 'url' and one optional attribute named 'mimetype'. It must not have any content or any attributes. If any other attributes or content do occur, implementations must ignore this. Specifically, the attribute named 'embedded' must not occur having any value other than an empty value, since this would make it an embedded last-image-source.

###### **3.1.1.1 uri**

This element holds the URI to the last opened image to use with this analysis-set. URIs are described in RFC 3986, located at <http://tools.ietf.org/html/rfc3986>.

###### **3.1.1.2 mimetype**

This element holds the mimetype of the source referred to in the url attribute. Mimetypes are described in RFC 2046, located at <http://tools.ietf.org/html/rfc2046>.

### **3.1.2 embedded last-image-source**

This form does not provide a reference to a source where an image is located. Instead, it simply provides an image.

An embedded last-image-source must have the attribute 'embedded'. Any value may be given to this attribute, but this value must be ignored by implementations.

Embedded last-image-source elements must also provide the attribute 'mimetype' holding the mimetype of their embedded data.

Any other attributes than embedded and mimetype must not occur in an embedded last-image-source. If they do occur, implementations must ignore them.

Finally, the content of an embedded last-image-source is the image-data itself, saved according to the specification for its mimetype and encoded into Base64. Base64 is described in RFC 3548, located at <http://www.faqs.org/rfcs/rfc3548.html>.

#### **3.1.2.1 embedded attribute**

This attribute signals that a last-image-source element is embedded. Its value is of no concern as long as it's not empty. The value itself must be ignored by implementations.

#### **3.1.2.2 mimetype attribute**

This element holds the mimetype of the embedded source. Mimetypes are described in RFC 2046, located at <http://tools.ietf.org/html/rfc2046>.

To maintain compatibility among implementations, the only mimetype that is currently supported is image/png. The PNG standard is described at <http://www.w3.org/TR/PNG/>. This means the content of the last-image-source element must be data for a PNG-image, encoded in Base64.

### **3.2 last-triangulation-source**

This element describes the source of the last triangulation used with the analysis-set. This information may be used by implementations to provide a quick way for users to continue working with the same triangulation with this analysis-set. The last-triangulation-source element is not required and must not occur more than once in a savefile element.

The last-triangulation-source comes in two forms. Implementations are recommended to implement both forms, but only the referring form is required. If the embedded form is not implemented, it must still be detected and disregarded when found while loading a savefile.

If a last-triangulation-source is found, but cannot be loaded due to restrictions such as access,

availability, recognition or any other reason, the last-triangulation element must be ignored.

### **3.2.1 referring last-triangulation-source**

This form provides a reference to a source where the triangulation is located. It has one required attribute named 'url' and one optional attribute named 'mimetype'. It must not have any content or any attributes. If any other attributes or content do occur, implementations must ignore this. Specifically, the attribute named 'embedded' must not occur having any value other than an empty value, since this would make it an embedded last-triangulation-source.

#### **3.2.1.1 uri**

This element holds the URI to the last opened image to use with this analysis-set. URIs are described in RFC 3986, located at <http://tools.ietf.org/html/rfc3986>.

#### **3.2.1.2 mimetype**

This element holds the mimetype of the source referred to in the url attribute. Mimetypes are described in RFC 2046, located at <http://tools.ietf.org/html/rfc2046>.

### **3.2.2 embedded last-triangulation-source**

This form does not provide a reference to a source where a triangulation is located. Instead, it simply provides a triangulation.

An embedded last-triangulation-source must have the attribute 'embedded'. Any value may be given to this attribute, but this value must be ignored by implementations.

Embedded last-triangulation-source elements must also provide the attribute 'mimetype' holding the mimetype of their embedded data.

Any other attributes than embedded and mimetype must not occur in an embedded last-triangulation-source. If they do occur, implementations must ignore them.

Finally, the content of an embedded last-triangulation-source is the triangulation-data itself, saved according to the specification for its mimetype and encoded into Base64. Base64 is described in RFC 3548, located at <http://www.faqs.org/rfcs/rfc3548.html>.

#### **3.2.2.1 embedded attribute**

This attribute signals that a last-image-source element is embedded. Its value is of no concern as long as it's not empty. The value itself must be ignored by implementations.

#### **3.2.2.2 mimetype attribute**

This element holds the mimetype of the embedded source. Mimetypes are described in RFC 2046, located at <http://tools.ietf.org/html/rfc2046>. Currently, no registered mimetypes are supported. The unregistered mimetype text/x-triangulation is supported and refers to the triangulation format as described in appendix A.

### **3.3 version attribute**

The version attribute of a savefile element describes the version of this specification that has been used to generate the information within that file. The value of this attribute must currently be '1.0'. If a higher versionnumber is encountered, but only the minor version number, i.e. the digit after the dot, has changed, implementations should try and read the savefile. Optionally they may warn the user and refuse. They must not silently fail. When the major version, i.e. the digit before the dot, has changed, implementations may silently fail to read the savefile.

Future versions of this specification will remain backwards compatible with all versions of this specification with the same major version number but a lower minor version number. They will also make a best effort to make sure older versions with the same major version number will remain forwards compatible, but no guarantees can be given due to possibly changing behaviour of plugins.

## **4. Analysis element**

This element describes an analysis as used in triAna. An analysis is a combination of plugins that modify the selection of geometry parts of a triangulation and plugins that provide output based on that selection, be it a visualisation of some sorts or other forms of output, such as printing, saving, special calculations or otherwise.

An analysis element is an element that only holds some children. These are plugin elements. No other content and no attributes must appear within an analysis element. If any do occur, these must be ignored by implementations.

### **4.1 plugin element**

The plugin element describes a plugin that was used in the analysis. This can be either a criteria plugin or an output plugin. It's up to the implementation to figure out what type it is. Because plugins are identified by an identifier and are rather generally set up, not much information is needed to save and load an analysis.

A plugin element must have exactly one attribute: an id attribute. No other attributes may occur in a plugin element. If other attributes do occur, implementations must ignore them.

The contents of a plugin element consist of zero or more setting elements. No other content may appear in a plugin element. If other content does appear in a plugin element, implementations must ignore this.

If one plugin element is incorrect, implementations must ignore that single element and may either continue loading the analysis without that single plugin, or may fail to load the complete analysis. The same holds when a plugin, for whatever reason, cannot be loaded or instantiated.

#### **4.1.1 id**

The id attribute holds the id of a plugin, as reported by that plugin. A plugin's id is not allowed to have any other characters than US-ASCII characters. Also, any form of quotes are prohibited. Hence the id will always be fit for inclusion in an attribute.

#### **4.1.2 setting element**

The setting element holds a single name/value pair associated with the plugin. The names of settings are derived from the property names of plugins.

A setting element must not have any attributes. If any attributes occur, these must be ignored.

The content of a setting element consists of exact two elements: one name element and one value element. No other content must be within a setting element. If any other content is within a setting element, implementations must ignore this.

##### **4.1.2.1 name element**

The name element holds the name of a property of the plugin. It must not have any attributes and its contents must be exactly the name of the property, without any extra whitespace around it. If any attributes do occur in the element, implementations must ignore them.

#### **4.1.2.2 value element**

The value element holds the value of a property of the plugin. The value element may have one attribute called 'encoding'. No other attributes must occur. If any other attribute do occur, implementations must ignore them. The content of the value element is the exact value of the property without any extra whitespace around it.

##### **4.1.2.2.1 encoding**

To ensure all types of values can be saved in the savefile, an encoding may be used for the value. If an encoding is used, this must be reported using this element. Only one such encoding may be used for the same value. Currently only Base64 is supported. Base64 is described in RFC 3548, located at <http://www.faqs.org/rfcs/rfc3548.html>.

## **Appendix A. Triangulation format**

The triangulation format developed by N. Hitschfeld is a simple, text-based format. The extension of files with contents as described in this appendix should have the extension '.tri'. Any streams with this contents as described in this appendix should be marked with mimetype 'text/x-triangulation'. See RFC 2046, located at <http://tools.ietf.org/html/rfc2046>, for more information about mimetypes.

### **Physical form**

Files or streams with triangulation data must be encoded in US-ASCII.

### **Compatibility issues**

Whitespace is important in this format. Specifically, newlines are important. Since newlines tend to give compatibility problems when looking at different operating systems, for the remainder of this appendix a newline character is understood to be a US-ASCII character with ordinal decimal value 10 or 13.

### **Comments**

Anywhere within a triangulation, a comment may appear, started by the single character '#', preceded by at least one whitespace character or at the beginning of the stream. Everything from this character up to next newline character, or the end of the stream if no such character exists, must be ignored.

### **Whitespace**

Whitespace is of great importance in this format, since it separates entities. Whitespace is understood in this appendix as being US-ASCII characters with ordinal decimal values 9, 10, 13 or 20.

Any whitespace that was not required or expected must be ignored by implementations.

### **Tokens**

To simplify the description all characters in the stream will be separated into tokens. A token is understood to be the longest consecutive array of non-whitespace characters starting at a character preceded by a whitespace character or at the beginning of the file. Comments must not be tokenized, but simply skipped when building tokens.

### **Numerical values**

Most information in the triangulation format is expressed in numerical values. Numerical values are tokens consisting of the characters '0', '1', '2', '3', '4', '5', '6', '7', '8' and '9'. Such a token must be interpreted as a decimal integer value. Optionally, a numerical value can start with a '-' character, making the value negative.

Floating point values are also possible. These values may include the character '.' one time. Such a token is to be interpreted as a decimal real value.



## **Vertices**

The first four tokens of the stream must be respectively “Numer”, “of”, “Vertices” and “=”. The fifth token of the stream is an integer numerical value, which is the number of vertices that will follow. For each vertex the stream must then have two tokens, each floating point numerical values, representing respectively x and y coordinates of the vertex. Vertices are indexed by their occurrence: the first vertex is numbered 0, the second vertex is numbered 1, etc.

## **Triangles**

After the vertices the first four tokens in the stream must be “Number”, “of”, “Triangles” and “=”. The next token of the stream must be an integer numerical value, which is the number of triangles that will follow. For each triangle the stream must then have three tokens, each integer numerical values, representing the indexes of the three vertices that make up the triangle.

Any tokens appearing after the triangles in the stream must be ignored by implementations.

## Appendix B. References

*Key words for use in RFCs to Indicate Requirement Levels*, S. Bradner, ed. IETF (Internet Engineering Task Force), March 1997. Available at <http://www.rfc-editor.org/rfc/rfc2119.txt>

*UTF-8, a transformation format of ISO 10646*, F. Yergeau, ed. IETF (Internet Engineering Task Force), November 2003. Available at <http://tools.ietf.org/html/rfc3629>

*Extensible Markup Language (XML) 1.0 (Fourth Edition)*, T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau eds. W3C (World Wide Web Consortium), 16 August 2006. Available at <http://www.w3.org/TR/2006/REC-xml-20060816/>

*Namespaces in XML*, Tim Bray, Dave Hollander, and Andrew Layman, eds. W3C (World Wide Web Consortium), 1999. Available at <http://www.w3.org/TR/REC-xml-names/>

*Uniform Resource Identifier (URI): Generic Syntax*, T. Berners-Lee, R. Fielding, L. Masinter, eds. IETF (Internet Engineering Task Force), January 2005. Available at <http://tools.ietf.org/html/rfc3986>

*Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, N. Freed, N. Borenstein, eds. IETF (Internet Engineering Task Force), November 1996. Available at <http://tools.ietf.org/html/rfc2046>

*The Base16, Base32, and Base64 Data Encodings*, S. Josefsson, ed. IETF (Internet Engineering Task Force), July 2003. Available at <http://www.faqs.org/rfcs/rfc3548.html>

*Portable Network Graphics (PNG) Specification*, T. Boutell, G. Randers-Pehrson, T. Lane, A. M. Costello, eds. W3C (World Wide Web Consortium), May 2004. Available at <http://www.w3.org/TR/PNG/>