MeshLab Documentation

Introduction

MeshLab is a advanced mesh processing system, for the automatic and user assisted editing, cleaning, filtering converting and rendering of large unstructured 3D triangular meshes. MeshLab is actively developed by the a small group of people at the Visual Computing Lab at the ISTI - CNR institute, a large group of university students and some great developers from the rest of the world. For the basic mesh processing tasks and for the internal data structures the system relies on the GPL VCG library.

Interacting with the mesh

When MeshLab starts you are in the so-called *camera mode* where the mouse actions are used to change the way in which you look at the model. This modality is the default and allows to easily navigate and inspect your model without actually modifying it. The camera mode is turned off when you use one of the Interactive editing tools, where the mouse actions are interpreted in a different way (e.g. for painting the mesh), but you can always switch back to the *camera mode* by pressing ESC; in this case the edinting tool is just suspended and you can go back to editing by pressing again ESC. To exit from an editing tool you have to press again the toolbar button that started it.

Standard camera mode	
left drag	**Rotate** orbit around the current center of the object. Click far from the center to rotate around the line of view.
ctrl+left drag	**Pan**
wheel	**Zoom**
shift+left drag	**Zoom** (if you do not have a mouse wheel or if you want smooth zoom)
ctrl+shift+left drag	**Light** rotate, to interactively change the default light setting.
left double click	**center and zoom** on clicked point; subsequent rotations and zooms will be centered on the chosen point.
alt+left drag	**Z translate**; moves the camera along the viewing direction, (the effect can be similar to zoom, but it allows to more precisely navigate in the space around the object)
shift+wheel	**change field of view** and move the camera to keep the model of approximately the same dimension (in practice increase/decrease the perspective deformation, much alike the Hitchcock zoom depicted in the Vertigo movie).
ctrl+wheel	**move near plane** back and forth to section the object and reveal its interior.
alt+wheel	**change point size** when rendering mode is Points. Use Points rendering to display interactively huge meshes.

Other useful key combinations for mesh inspecting are:

- **ctrl+d** toggle double sided lighting
- **ctrl+k** toggle backface culling
- **ctrl+f** toggle blue-red back=front lighting

Obviously for mac users, you should substitute the ctrl with the command key.
With one of the two selection tool enabled Click = click left button of the mouse

click	**Select** all the faces in the direction (including the hidden ones behind)
ctrl+click	**Add** faces to selection
shift+click	**Remove** faces to selection
alt+click	**Select** only the front faces (without the hidden ones behind)
ctrl+alt+click	**Add** only the front faces (without the hidden ones behind) to selection
shift+alt+click	**Remove** only the front faces (without the hidden ones behind) to selection
wheel	**Zoom**
middle mouse button	**Translate** mesh
alt+ middle mouse button	**Rotate** mesh

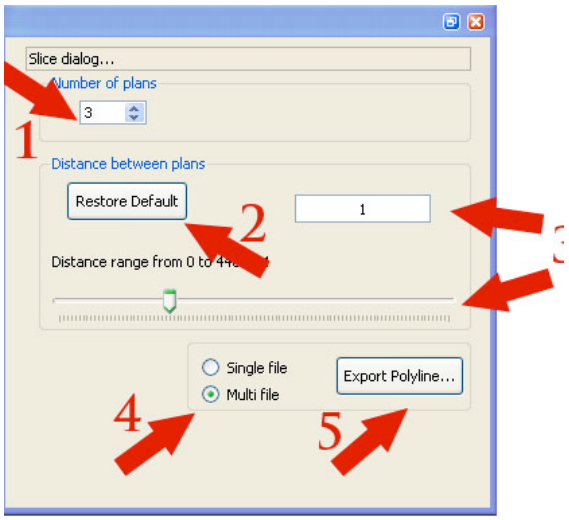
Interactive editing tools

You can access to these tools by activating them from the *edit* menu or from the toolbar (all the buttons on the right). Only one of these tools can be activated at a time. When you have an active editing tools all the other tools are disabled. Activating one of these tools usually changes the way you interact with the mesh: mouse actions are directly processed by the chosen tool. For example when you start to use the Painting Tool, your mouse drags are interpreted as brush strokes over the mesh. Remember that you can toggle back and forth between the standard *camera* moving mouse mode and the current tool mode by pressing esc. Switching to camera mode does not disable the tool. To exit from the editing mode press again the same tool button. You can see that you have completely closed an editing mode by seeing again as active all the other editing buttons.

Plane Slicing Tool

The **Editing Plane Slicing tool** allows users to create one or more mesh's slices obtained through the intersection of a set of parallels plans with the mesh. The semi-transparent plans are painted in the main windows, centers to the object's bounding box that we are modifying, with simplicity users can found the exactly position of the plans through the geometry; the editing of the plans are not bound to the rest of the scene.

(Picture 1.) Editing dialog



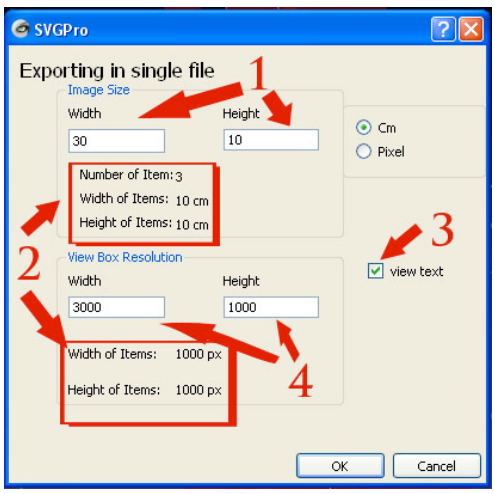
The tool include a dialog (Picture 1) with the function:

- **1** Define the number of plans.
- **2** Restore the default settings.
- **3** Increase/decrease distance with a slide or editing manually.
- **4** Single/multi file define the exporting mode
    - **Sigle File Mode** Single file dispose the result of the intersections over a single SVG file, in a continuous strip, each item designed inside of a rectangle.
    - **Multi File Mode** Each item has its file, the x-th slice named namefile_X.svg has a single slice.
- **5** Start with exporting function.

The intersection result could be exported in Scalable Vector Graphics (SVG), there are two possible modes:

- **Single files**, if the plans are more than one all the result are stored in a single file.
- **Multi files**, for each plans are created a file.

(Picture 2.) SVG properties dialog.



The dialog in (picture 2) contains the main properties of the SVG document:

- **1** Dimensions in centimeters/pixels of the global context
- **2** A dinamic set of label that calculate each dimension's item, particularly useful in multifile exporting.
- **3** Enable/disable details and text inside the context.
- **4** Resolution of the global context in pixels.

Alignment

Alignment is a rather complicate process. First of all you should have well understood the layers mechanism in MeshLab and the fact that each mesh can have a transformation matrix. The alignment process simply modifies the transformation of each layer. The main idea is that you iteratively glue your misaligned meshes over those already aligned. A mesh that is aligned together with a set already aligned mesh is said to be *glued* and a * is shown near to its name. Initially all meshes are unglued. You task is to roughly align all your meshes. Once the meshes are glued in a rather good initial position you can start the alignment process: the system chooses what meshes have some overlapping part and for each pair of meshes the system starts a ICP alignment algorithm that precisely aligns the chosen pair. At the end of the process all the glued meshes will hopefully be aligned together.

Key Concepts:

- ICP: Iterated closed point: The basic algorithm that automatically precisely aligns a moving mesh M with a fixed one F. The main idea is that we choose a set of (well distributed) points over M and we search on F the corresponding nearest points. These pairs are used to find the best rigid transformation that bring the points of M onto the corresponding points on F. ICP has a lot of tunable parameters.
- Global Alignment: also known as multiview registration. A final step that evenly distributes the alignment error among all the alignments in order to avoid the biased accumulation of error.
- Absolute Scale. The parameters of the alignment tool are in **absolute** units, and the defaults are acceptable for a standard scanner outputting meshes in millimeter units. So for example the target error (i.e. the error that the ICP try to achieve) is 0.05 mm something that can be achieved with a good scanner. Obviously if your range maps are in a different unit (microns, kilometers, etc.) you must adjust the default alignment parameters or the alignment process will fail.

Small Tutorial

1. Open all the meshes provided in the SampleRangeMaps package, by simply selecting all the mesh in the open dialog (use ctrl for multiple selection as standard). All the meshes will be loaded together in different layers.
2. Start the alignment tools (the *A* in the toolbar)
3. Choose a mesh and glue in its current position (a star will appear beside its name).

4. Choose another mesh (not glued) and glue it in the correct position by starting the *point based alignment*. A new window will popup and you have to choose four corresponding pair of points on the two meshes (the set of the already glued one and the new unglued mesh). You choose points by double clicking over the two meshes.
5. Repeat step 4 until all the meshes have been glued.
6. Click on "Start process." Now the system will search for overlapping meshes and for all the meshes that have some overlaps compute a fine ICP. At the end of this process a "Global Alignment" step will distribute residual error evenly.
7. Now the alignment is complete and you can look at results and check for any errors in the alignment process.
8. Close the Align Tool.
9. Flatten all the layers into a single mesh.
10. Start the merge process by using a surface reconstruction algorithm like the Poisson based one.
11. Save to a new file.

The editing **Quality Mapping Tool** is a tool for the fine tuning of how scalar values, defined over a mesh, are mapped into colors. Note that this filter will not work if quality values are not defined over the mesh vertexes. The tool is composed by two main section:

Transfer Function

The Transfer Function maps, through a linear chart, the distribution of the values of RGB channels over a linear space, identified by a color band. The transfer function view is interactive, in fact the user can modify the distribution of the channels values using handles, that can be added, moved or removed. To add a new handle to the Transfer function, it's necessary first to select the channel for the handle (by selecting the forward channel in the top-right radio button group, or by clicking on an handle of the same channel), and then double-clicking in an empty place of the transfer function view. It's possible to remove an existing handle by double-clicking over it. Every time a change is performed to the transfer function, it's immediately visible in the color band below. More over, it's possible to save the current transfer function in a CSV file or load it. Important Note: When a transfer function is loaded from an external file the changes to the current transfer function will be lost.

Equalizer

It represents, through an histogram chart, the distribution of the quality over the mesh (x-axis represents quality levels, y-axis represents the number of vertexes with a certain quality). Like the transfer function view, the histogram one too is provided of handles, exactly three. Each handle has a different meaning: the leftmost one, represents the minimum quality limit, the rightmost one the maximum quality limit, and the middle one represents the position of the middle quality level respect to the minimum and the maximum actual values. The portion of histogram delimited by lateral handles is mapped in the transfer

function (visible as background of the TF view). So, all the quality levels included in the range between the histogram beginning and the left handle position will be mapped with the leftmost color of the color band, while the ones between the right handle and the histogram end will be mapped with the rightmost color. All the other values of the histogram are mapped in the color band depending by the relative position of the middle quality value. This value is the parameter of the exponential (or logarithmic) function used to interpolate the TF nodes when mapping quality levels on color band. The curve representing this function is viewed in the gamma correction panel. If a lateral handle is moved outside the histogram, or if the user sets in a "lateral" spin box a quality value greater than max or lesser the min, the histogram will auto stretch itself. Finally a slider can be used to change the brightness of the color band colors.

Tutorials

Stressing the curvature of a mesh

1. Apply the filter Colorize/Gaussian Curvature (equalized)
2. Open the Quality Mapping Tool
3. Choose the RGB color preset
4. Quality values over Gamma Curvature are often concentrated in a tight range between -*a* and +*b*. So manually set middle quality to 0. Try setting minimum and maximum equalizer values to stretch the significant part of the histogram all over the color band, so that negative values will be colored in red and positive ones will be colored in blue, while quality values around 0 will be colored in green
5. You can even modify the transfer function to eliminate the green color and extend red and blue to all the color band

Giving a tone to a mesh

1. Apply the filter Colorize/Ambient Occlusion
2. Open the Quality Mapping Tool
3. Choose a color preset (i.e. Red Scale)
4. Modify the transfer function to obtain the color scale you like
5. Set the minimum/middle/maximum quality value in equalizer to adjust the tone depending on the distribution of quality over the model

Applying the same mapping to several meshes

1. Set the Quality Mapping tool with significant values
2. Save preset to a file (it will contain transfer function and equalizer parameters) and close the tool
3. Open a new mesh
4. Open Quality Mapping tool
5. Click on load preset,locate the CSV file you previously saved and apply the filter.

Alternatively open the filter Quality Mapper Applier, locate CSV file and apply the filter. This filter can be used also in Meshlab command line. Note: all the files should have similar quality values to be colored in a significant way(i.e. applying the same colorize filter)

Straightener

The **Editing Straightener Tool** is a tool for interactively change the transformation matrix associated with every loaded mesh.

Note: This tool is **not** intended for generic transformations like scaling, shearing or mirroring. Such non-repositioning transformations can be achieved using the "Apply Transform" filter.

Features

This tool offers several ways to change mesh alignment and origin:

- Buttons for flipping on, swapping between and rotating around all the primary axes.
- A button to align the axes with the current view.
- Buttons and sliders to put the coordinate origin in any point inside the mesh bounding box.
- A button to put the coordinate origin in the trackball center.
- Interactive "freehand" modes:
    - User can drag the mesh wrt the coordinate frame.
    - User can drag the coordinate frame wrt the mesh, using a special eulerian trackball, also able to make angle-constrained rotations.
- Ways to literally "draw" one or two axis direction directly on the mesh.
- Ways to align the coordinate frame on the best fitting plane of the eventually selected vertexes in the mesh.
- A button to freeze the current transformation in the mesh.

The tool also offers an integrated undo system and control over several visual feedbacks.

Sample Usage

Here we show, through examples, several ways to use this tool:

Basics

- Open the file "screwdriver.ply" in the "sample" directory.

This mesh's default orientation is such that if the view trackball isn't rotated, the mesh itself is hardly recognizable.

- Click on the "Straighten up a mesh" button.

Now the tool user interface is shown, and dragging the mouse on the mesh will have no effect.

- Click on the Swap Z with X button, then on the Rotate 90 degrees around X button

Now the mesh transformation matrix has been changed, but the mesh has not been modified.

- Click on the freeze button

Now the mesh has been updated, and the new orientation can be saved.
Like every other editing tool, Straightener can be suspended using the "Not editing" button, and the view trackball can be modified.

Simple Straightening

- Open the file "sub.obj" in the "sample" directory and start the tool.

This mesh's default orientation is plain wrong, this can be easily seen from the visual feedback.

- Suspend the tool (esc) and rotate the view trackball until the submarine side is correctly viewed.
- Resume the tool (esc again) and click on the "align with actual view" button, then click on center on bbox button (in the origin tab).

Now the mesh is correctly orientated, and can be freezed.
The coordinate frame orientation and position can be refined, toggling the "Freehand mesh dragging" button:

- Left click rotates around the current axis
- Right click tranlsates along the current axis
- The default current axis is x, it can be changed to y and z using the ctrl and shift keyboard modifiers (visual feedback is provided for ease of use).
- Rotation can be done in a controlled fashion setting the snap value.

The new coordinate frame can be accepted toggling the "Freehand mesh dragging" button again.

Layer Composition

This tool can be used to compose multiple layers.

- Open the file "twirl.off" in the "sample" directory and start the tool.
- Click on the "flip y and z" button, freeze and close the tool.
- "Open inside" the mesh the file "sharp sphere" and restart the tool.

Note that the tool works always in the layer active *when the tool started*.

- Click on the rotate 90 around z button, then toggle "Freehand mesh dragging" (in the Drag tab). A blue wireframe sharp sphere is now rendered over the original one, that can be dragged like the view trackball (without scaling, of course).
- Ctrl-drag the wireframe sphere to pan it up the top of the twirl. Again, the tool can be suspended and the view changed during all these operation, if necessary.
- Toggle back the "Freehand mesh dragging" button to accept the new sphere position, then freeze and close the tool.

Now the composed layers can be merged/flattened down to a pawn-resembling mesh.

Special Features

This tool has also two families of special features, useful if the mesh is not straight and not regular, but has some regular surfaces that can be taken as reference (e.g. the basement of a scanned statue).
The first family is the "draw axes on mesh" one: using one or two "rubberbands" (similiar to the Measuring Tool one) the user can draw a segment or two on the mesh surface, and the coordinate system will rotate around its origin to align the chosen first axis to the first drawn segment. Then, if another axis has been drawn, the coordinate frame will rotate *around the first axis* to align the second one as much as possible.

The second family is the "get from selection", and is available only if some vertexes are selected (obviously). From the selection a best fitting plane is computed, then the chosen coordinate frame plane is aligned with the computed one, and the origin is put on the plane projection of the selection bounding box center

Point Picking Tool

Description
This tool can basically be described as a mesh labeling tool. It allows you to pick and name points on a mesh and save the output to xml.

General
The tool has two main modes, normal and template mode. Template mode is described below. There are two main modes of interaction with the points, **Pick Point**, **Move Point**, and **Select Point** mode. The mode of interaction is indicated by the radio at the top. In pick mode you add new points to the mesh by right clicking on the mesh. In move mode you can move a point by right clicking near the point and dragging it to a different location. In select mode you can right click a point on the mesh and it will be highlighted in the pick points edit tool menu.

Here are some notes about the **active** checkbox. The immediate affect of checking the active checkbox is to hide/show the picked point on the mesh. This feature is meant to allow to use a template for which scan data will have one of the template points missing. In this case you can unflag this point, labeling it in-active. When points are added you have a green cone, representing the normal, with a label next to it. You can also choose at the bottom where it says **Normal Options** to have just a dot, which is the point, a label for that point, or a dot, label, and a green line which represents the normal of the face the point is on.

Template Mode
Summary: Templates are use to facilitate the labeling of many meshes with the same labels. If you have a template loaded you are in Template Mode. The name of the template will be written in blue letters next to the place at the bottom where it says "Template Name:". "No Template Loaded" will be printed if no template is loaded.

In Template mode you will only be able to pick and move the points that are already defined as part of the template. Use the **Add Point** and **Remove Point** buttons in the Template Controls area to modify a template. You can also start from scratch, pick some points, then save them as a template (saving the names so you can use them for other meshes).

Saving/Loading Points
Points are saved to **a xml file** by using the **save button** at the top. Points can be loaded from a file by pressing the load button. Alternately points are **automatically loaded from a file** if you had previously saved them using the filename recommended by the tool.

Points are **saved to PerMeshAttributes** automatically when you close the edit plugin and loaded back from PerMeshAttribute when the plugin is loaded. This takes precedent over loading points from a file.

Other Features

1. Rename a point by slowly double clicking on it
2. Undo: will undo only that last move

Use With "Align Mesh using Picked Points" filter

In order to use the **Align Mesh using Picked Points** filter you need to have either saved points:

1. to PerMeshAttributes or
2. to a file with the name suggested by the point picking plugin.

Note: that points are only needed if "Use Markers for Alignment" is checked.
Note: that if "Scale the mesh" is checked then you need points and to have checked "Use Markers for Alignment".
Note: the transform calculated is stored in per mesh data with key AlignTools::getKey()

Notes for Developers who want to use picked points in their plugin

You can look at how the filter FP_ALIGN_WITH_PICKED_POINTS in the cleanfilter plugin uses points picked by the edit_pickpoints plugin. Note, most code is here, cleanfilter/cleanfilter.cpp but the important bit is here:

```
MeshModel *stuckModel = par.getMesh(StuckMesh); PickedPoints *stuckPickedPoints = 0; ... //first try to get points from memory if(vcg::tri::HasPerMeshAttribute(stuckModel->cm, PickedPoints::Key) ) {
        CMeshO::PerMeshAttributeHandle<PickedPoints*> ppHandle = vcg::tri::Allocator<CMeshO>::GetPerMeshAttribute<PickedPoints*>(stuckModel->cm, PickedPoints::Key); stuckPickedPoints = ppHandle();
```

What you get is a PickedPoints object. To see what it does look here /meshlabplugins/edit_pickpoints/pickedPoints.h(cpp).

Basically what you need to know is that this class hold a std::vector<PickedPoint*> where a PickedPoint is a simple object that has two members:

1. name of point: QString name;
2. point: vcg::Point3f point;

Other information

- source code in edit_pickpoints folder

Hole Filler Tool

The Interactive Hole Filler Tool is a tool allow the user select holes and edit them in different ways. The goal feature of this plugin is fill holes as user want, but to do this it's also possible edit hole's topology.

Prerequisities
Hole Filler Tool need meshes manifoldness.

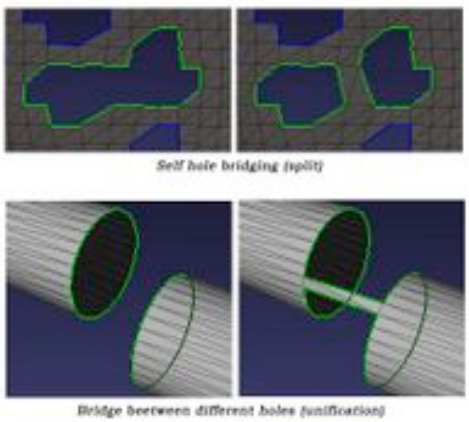
Hole Selection:
User can select an hole to edit with a mouse click over a border face or pinning the checkbox in dialog.

Hole Filling:
Basic filling algorithm use a tecnique wich adds a face between two adjacent border edge (friendly called "ear"). This is a greedy algorithm which choose each time the best pair of adjacent border edge into the hole. However selected holes can be filled follow different criteria:

- *Trivial*: best ear is computed only with face (triangle) quality
- *MinimumWeight*: best ear is computed mixing quality face and angle beetween ear and adjacent faces.
- *SelfIntersection*: as Minimumweight but pay attention to autocompenetrating face.

When holes are filled user can fill again, maybe with different params, or accept or discard patch applied by filling. As selection, acceptance could be done clicking the mesh or directly into the dialog.

Hole Bridging:



Bridges type
Sometimes right filling must be driven by user, to do that it's possible add some bridge, manually or automatically. Bridge consist into 2 adjacent face which connect 2 different border edge. With bridge an user can:

- split an hole, if bridge is builded with starting and final border edge belong to same hole
- unify more holes into one, if bridges connect edges belong different holes.

Manual Bridging:
User select the two edges to connect picking the border face. It cannot build a bridge beetween 2 adjacent edges.

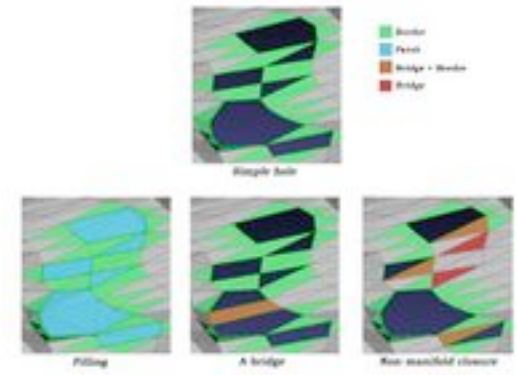
Automatic bridging:
Automatic briding works on selected holes. User have to specify if he want connect all selected holes or split each selected hole.

Non Manifold Hole Splitting:
Filling can manage non manifold holes, but this plugin allow edit hole structure to split an non manifold hole into many manifold ones.

Data Structures

- *FgtHole*: Extend vcg::tri::Hole<MESH>::Info adding some information like the hole status (selection, autocompenetration, filling...). Also add functionality to draw, fill and restore...
- *FgtBridgeBase*: abstract class which exposes method used by HoleSetManager.
    - *FgtBridge*: bridges used to connect 2 border edge. It can split an hole or unify 2 (or more) holes.
    - *FgtNMBridge*: faces added to close non manifold holes.
- *HoleSetManager*: it is an unique entity to manage the holes and the bridges existing into the same mesh (so with the same additional data).
- *HoleListModel*: model of model/view architecture used by gui.

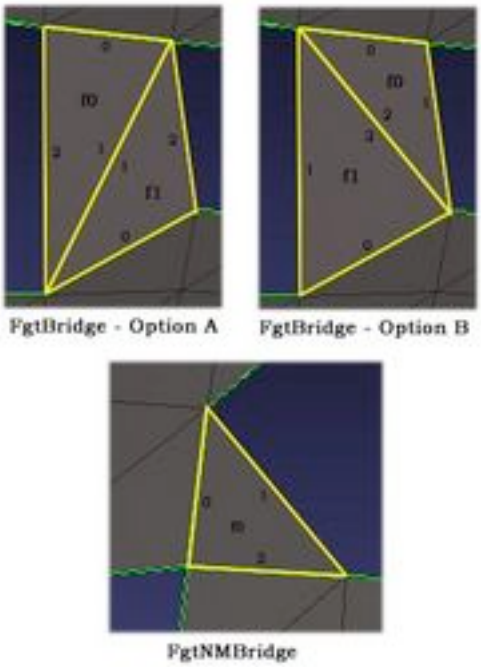
Mark interesting faces



Flags usage
Interactive Hole Filler tool mark interesting faces using additional data. This marker, with holes properties, are used to know topology of holes and its filling.
Characteristic face types are:

- *HoleBorderFace*: marks faces which have at least an edge adjacent to hole, border faces or faces adjacent to hole's filling
- *HolePatchFace*: marks faces added to fill hole
- *PatchCompFace*: marks faces self-intersecting with mesh
- *BridgeFace*: marks faces added to edit holes

Bridges bulding issue



Topology of bridge's face

FgtBridge

Rappresent the real meaning of word bridge. It connects 2 edge, has 2 abutment over the mesh and has 2 borer side. It is a pair of faces, both adjacent to each other. There is 2 way to triangulate 2 edges (4 vertex) so it'll choose configuration with best quality of its face. Bridge is build to have edge 0 adjacent to mesh for both its faces, and its faces share edge 1 or 2 following the triangulate configuration.

FgtNMBridge

It's a single face used to change non-manifold vertex in manifold one. It has vertex 0 on non manifold vertex and edge 0 and 2 adjacent mesh. Its edge 1 could be a border edge of another hole, but it could fill entirely the sub-hole (if sub hole has only 3 brder edge).

Slicing a mesh

Version 1.1.1 of MeshLab can do slicing. It is a bit (!) buggy, and with a rather clumsy interface but more or less works. Next version will have a cleaner more robust interface (hopefully). Slices are exported in svg format. Conversion from 2D svg to 2D dxf can be done by inkscape.

micro tutorial

1. Open a mesh
2. rotate it with the transformation dialog so that slicing with planes parallel to the YZ plane is ok (use the render-> show axis for checking)
3. start the slicing tool

4. dragging in the window move the slicing plane orientation, so DO NOT drag over the window if the yz plane is ok for you. If you want to examine the model press the trackball icon in the tool bar.
5. set the desired number of planes
6. press export (you can choose if you want to have all your slice in single file or in multiple files)
7. you can customize a bit the output, by removing label, deciding the size of the output.

Cleaning a Mesh

Cleaning a mesh means trying to remove all the small geometrical and topological inconsistencies that can be found in off-the shelf meshes. Common problems that usually have to be checked are the following ones:

- Duplicated vertices
- Unreferenced vertices
- Null Faces
- Self Intersecting faces
- Non Manifold faces
- Small Holes. Small holes (small in terms of number of the edges that compose their boundary) can be safely autmatically closed. Larger holes needs some special care...
- Small dangling artifacts. It happens that after a 3D scanning there are a lot of small *pieces* floating in the air, unconnected with the rest of the surface. These pieces are *small connected components*; you can remove them with the apposite filters, that can discriminate according to the size of the pieces in terms of faces composing them or their spatial diameter.
- Removing Noise. This is a more complicated task. This happens usually when managing 3D scanned meshes. It happens that surface that you expect to be perfetly flat are affected by some kind of noise and are irregular. Just like for image processing noise removal has positive and negative sides. MeshLab supports differnt kind of smoothing filters that are able to remove various kind of noise:
    - Laplacian
    - HC Laplacian
    - Two Step Smoothing
- **Duplicated vertices**
- If your model is coming directly from a 3D scanner, there will be many duplicate definitions for each vertex. If the model is manipulated in MeshLab and then exported to another software package, these duplicate points can cause problems with rendering and conversion to different formats. By removing these duplicate vertex definitions, the model can be simplified and downstream software problems reduced.
- A small discussion on duplicated vertices, unreferenced vertices and how to remove them with MeshLab can be found in the MeshLab stuff blog
- Open your model
- The message bar at the bottom of the screen will indicate the number of faces and the number of vertices in the model

- A dialog will pop up asking if you want to unify duplicated vertices. Select No.
- After the model loads, click on the Filters Tab
- Select the Clean submenu.
- Select Remove Duplicated Vertex
- After performing this cleaning step, there should be a significant decrease in the number of vertices associated with the model.

Image-based 3D Reconstruction

Meshlab, in conjunction with another tool called *Epoch 3D Webservice* is able to produce accurate 3D model starting from a set of images of the object to reconstruct. The steps to follow are:

1. Process your images with the Epoch 3D Webservice
2. Import the output of the Epoch 3D Webservice with Meshlab
3. Merge the range maps and build the model

Merge the range maps and build the model

Here we describe the various parameters involved in the creation of range maps from the raw data coming out from the web reconstruction service. It is possible to modify several parameters in order to produce 3D range maps that are cleaner and affected by lower noise.

Parameters description:

- **Subsample**
  Indicates the level of sub-sampling of the range maps used. Example: for a range map of 600x400 a level of 2 indicates that you use a 300x200 pixels in total for each range map. This parameter is used to reduce the final number of vertices of the model. The reduction is done in a weighted manner.
- **Minimum Count**
- Indicates the minimum number of occurrences required for a value of depth to be considered for the reconstruction. Example: if *Minimum Count* is set to 6 a value of depth that has no correspondences in at least 6 range maps is not used to build the model.
- **Minimum Angle**
- faces that haves their normal forming an angle with view direction larger than the indicated value (in degree) are deleted.
- **Remove Pieces less than**
- After the meshing all the dangling pieces composed by a number of faces smaller than the indicated values are deleted. In this way you can automatically delete all the small floating pieces that usually are just noise.
- **Depth Filter**
- The depth filter is used to reduce the noise of the estimated range maps. Sometimes could happen that the computed range maps are noisy, especially near high depth gradient. With this parameters the user can control a filter bank (*dilation* and *erosion* filter) to reduce this undesired

effect that results in geometric artifacts on the model's surface. The dilation filter is applied before the erosion one. The defaults parameters are good in most cases. The erosion reduce the reconstructed surface area, hence to reconstruct more detail reduce the steps or size of this filter. If the range maps still remains noisy after the filtering increase the erosion steps or size. Increase dilation steps or size in case of holes in the reconstructed surfaces.

- **Fast Merge**
- This option active a fast merge process but produce a less accurate model. If high accuracy is not required or many range maps have to be processed this option should be active.

Layers

Since version 1.1 MeshLab supports the management of meshes organized into multiple layers. In simple words it is possible to have many meshes in the same window. Each mesh is kept on a separate layer and most of the tools/filters are designed to operate to only one of them at a time (for example smoothing filters smooths out only the current layer). Saving and loading are usually single layer oriented too (e.g. when you save a ply you save just the current layer). As a reasonable metaphor of the layer inside meshlab you can consider how the layer mechanism works in a bitmap editing application (like gimp).
A set of layer build up a *project* that can be saved a separate entity (in the simple ALN ascii format).
Each layer stores also a 4x4 transformation matrix. This transformation matrix is used mostly for rendering and has a leading role in the [[alignment] process where it codes the relative transformation between the various range maps. All the processing filters normally works in the *natural* space of the mesh, with the plain exception of filters specifically designed to work with these transformation, like the *transformation* filter and the *freeze* filter that make this transformation perpetual actually changing the coordinates of the mesh vertexes.

To have multiple active layers you can alternatively:

- select more than a single file in the open dialog and press ok.
- open a mesh and then use the 'open as a new layer' command in the file menu
- open a mesh, show the layer dialog (button on the toolbar) and press the "+" button.

You can merge different layers together by means of the 'flatten' command. The flattening operation simply put the mesh together in the same layers freezing their coordinates.

Layer Managment

Meshlabs has a powerful Layer Managment Tools. It is similar to other softwares like Adobe Photoshop, The Gimp, etc. To manage different layers, press the 5th button in the task bar (some gray planes). A tab opens at the left side. Here you see all your layers.

Be careful, when you open a new mesh, the layers box remains unchanged - you have to open it again to have the correct layers.

Managing the Layers
You can add an external mesh to a new layer by going in File > Open As New Layer. To manage the layers, you can either right-click on the layer box, or go in Filters > Layers Management.

Then you have 5 options :

1. Automatic Pair Alignment -> ?
2. Freeze Current Matrix -> ?
3. Move Selection to another Layer -> Create a new layer with the selected faces. Select some faces on your model, then choose "Move selection to another layer". You have a very explicit option "Delete original selection" : you either *move* or *duplicate* your selection on the new layer.
4. Duplicate Layer -> Create a new layer with the same mesh as the selected layer. The difference with the option above is that it copys *all* the mesh, not only the selected faces.
5. Flatten visible Layers -> You can merged different visible layers. The options are again explicit.

Interacting with the layers
You can work with only one layer at each moment, the one which is selected in yellow. This includes saving (**/!\only the selected layers is saved/!**), selection, and more globally all interactions. To choose the layer with whom you want to work, simply left click on it in the layer box. You can also hide or print each layer with the small "eye" icon.

Tip : Something very powerful is that you can select faces on a layer, then hide it (this layers being still selected). Doing so allows you to see the selected faces in red, but transparent - it is quite useful to compare meshes, by example.

Transforming a Mesh
To transform a Mesh (translation, rotation, scale), go in Filters > Normals, Curvature and Orientation > Apply Transform. There you have 3 tabs which allows you to translate, rotate and scale your mesh.

Tip : Think to show the axis (Render > Show Axis), it really helps :)

When you change the parameters, you can either choose : -Absolute option : clear the previous transformation and apply only the current set -Relative option : include the current set to the previous transformation. For each transformation press "Apply" to see what happens.

Translation Very simple to use, you can set a direction in X, Y or Z. The axis helps you to know the direction. The length values are not absolute : if you apply a transformation, the length unit can change. Useful option *Move Center To Origin*

Scale Very simple to use too, you can change the scale in X, Y, or Z. Useful option *Uniform* to set the same scale in the 3 directions. Be careful : if you combine this option with rotation, the results might be surprising sometimes.

RotationThe most difficult option. You can set a rotation around an axe (X,Y, Z) in degrees. The axis helps you too to know the direction. You have 2 options : turn around the center of the mesh (center of gravity ?) and turn around the origin. But there is a trap (:p) : *the origin is not located in the intersection of the axis !* It is located on the origin of the mesh, coordinates (0,0,0), which is different of the origin of the shown axis. When you are satisfied with your transformation, press "Close and Freeze". Warning : you cannot undo it ! And you lost the transformation matrix associated, so think to copy it before if you need it ! "Close" cancels everything.

Tip : if you have strange behaviors with rotation or scale, close and freeze the transform, then open the filters again. You may be in extreme values, close to the precision levels.

Filters

The Filter List page presents a simple long list of all the filters that will be available on the 1.2. version.

- Simplification
- Meshing
- Align Mesh using Picked Points
- Morph Mesh
- MLS Surfaces
- Convex Hull/Alpha Shape/Voronoi Reconstruction filter

Filter List 1.2.0

- ***Aging Simulation*** Simulates the aging effects due to small collisions or various chipping events
- ***Vertex Ambient Occlusion*** Generates environment occlusions values for the loaded mesh
- ***Face Ambient Occlusion*** Generates environment occlusions values for the loaded mesh
- ***Automatic pair Alignment*** Automatic Rough Alignment of two meshes. Based on the paper **4-Points Congruent Sets for Robust Pairwise Surface Registration**, by Aiger,Mitra, Cohen-Or. Siggraph 2008
- ***Ball Pivoting Surface Reconstruction*** Reconstruct a surface using the **Ball Pivoting Algorithm** (Bernardini et al. 1999). Starting with a seed triangle, the BPA algorithm pivots a ball around an edge (i.e. it revolves around the edge while keeping in contact with the edge endpoints) until it touches another point, forming another triangle. The process continues until all reachable edges have been tried.
- ***Remove vertices wrt quality*** Remove all the vertices with a quality lower smaller than the specified constant
- ***Remove isolated pieces (wrt face num)*** Remove isolated connected components composed by a limited number of triangles

- ***Remove isolated pieces (wrt diameter)*** Remove isolated connected components whose diameter is smaller than the specified constant
- ***Align Mesh using Picked Points*** Align this mesh with another that has corresponding picked points.
- ***Select Faces by view angle*** Select faces according to the angle between their normal and the view direction. It is used in range map processing to select and delete steep faces parallel to viewdirection
- ***Remove T-Vertices by edge flip*** Removes t-vertices by flipping the opposite edge on the degenerate face if the triangulation quality improves
- ***Remove T-Vertices by edge collapse*** Removes t-vertices from the mesh by collapsing the shortest of the incident edges
- ***Remove Duplicate Faces*** Remove all the duplicate faces. Two faces are considered equal if they are composed by the same set of verticies, regardless of the order of the vertices.
- ***Merge Close Vertices*** Merge togheter all the vertices that are nearer than the speicified threshold. Like a unify duplicated vertices but with some tolerance.
- ***Colorize by Quality*** Colorize vertex and faces depending on quality field (manually equalized).
- ***Discrete Curvatures*** Colorize according to various discrete curvature computed as described in: '*Discrete Differential-Geometry Operators for Triangulated 2-Manifolds*' M. Meyer, M. Desbrun, P. Schroder, A. H. Barr
- ***Triangle quality*** Colorize faces depending on triangle quality:
  1: minimum ratio height/edge among the edges
  2: ratio between radii of incenter and circumcenter
  3: 2*sqrt(a, b)/(a+b), a, b the eigenvalues of M^tM, M transform triangle into equilateral
- ***Self Intersecting Faces*** Select only self intersecting faces.
- ***Self Intersections*** Colorize only self intersecting faces.
- ***Border*** Colorize only border edges.
- ***Texture Border*** Colorize only border edges.
- ***Color non Manifold Faces*** Colorize the non manifold edges, eg the edges where there are more than two incident faces
- ***Color non Manifold Vertices*** Colorize only non manifold edges eg.
- ***Laplacian Smooth Vertex Color*** Laplacian Smooth Vertex Color
- ***Laplacian Smooth Face Color*** Laplacian Smooth Face Color
- ***Vertex to Face color transfer*** Vertex to Face color transfer
- ***Face to Vertex color transfer*** Face to Vertex color transfer
- ***Texture to Vertex color transfer*** Texture to Vertex color transfer
- ***Random Face Color*** Colorize Faces randomly. If internal edges are present they are used
- ***Vertex Color Filling*** Fills the color of the vertexes of the mesh with a color choosed by the user.
- ***Vertex Color Invert*** Inverts the colors of the vertexes of the mesh.
- ***Vertex Color Thresholding*** Reduces the color the vertexes of the mesh to two colors according to a threshold.
- ***Vertex Color Brightness*** Change the color the vertexes of the mesh adjusting the overall brightness of the mesh.

- ***Vertex Color Contrast*** Change the color the vertexes of the mesh adjusting the contrast of the mesh.
- ***Vertex Color Contrast and Brightness*** Change the color the vertexes of the mesh adjusting both brightness and contrast of the mesh.
- ***Vertex Color Gamma Correction*** Provides standard gamma correction for adjusting the color the vertexes of the mesh.
- ***Vertex Color Levels Adjustment*** The filter allows adjustment of color levels. It is a custom way to map an interval of color into another one. The user can set the input minimum and maximum levels, gamma and the output minimum and maximum levels (many tools call them respectively input black point, white point, gray point, output black point and white point).
- ***Vertex Color Colourisation*** Allows the application of a color to the mesh. In spite of the Fill operation, the color is blended with the mesh according to a given intensity. .
- ***Vertex Color Desaturation*** The filter desaturates the colors of the mesh. This provides a simple way to convert a mesh in gray tones. The user can choose the desaturation method to apply; they are based on Lightness, Luminosity and Average.
- ***Equalize Vertex Color*** The filter equalizes the colors histogram. It is a kind of automatic regulation of contrast; the colors histogram is expanded to fit all the range of colors.
- ***Vertex Color White Balance*** The filter provides a standard white balance transformation. It is done correcting the RGB channels with a factor such that, the brighter color in the mesh, that is supposed to be white, becomes really white.
- ***Box*** Create a Box
- ***Sphere*** Create a Sphere
- ***Icosahedron*** Create an Icosahedron
- ***Dodecahedron*** Create an Dodecahedron
- ***Tetrahedron*** Create a Tetrahedron
- ***Octahedron*** Create an Octahedron
- ***Cone*** Create a Cone
- ***Conditional Vertex Selection*** Boolean function using muparser lib to perform vertex selection over current mesh. It's possible to use parenthesis, per-vertex variables and boolean operator: **(,)**,**and**,**or**,**<>**,**=**
  It's possible to use the following per-vertex variables in the expression: x, y, z, nx, ny, nz (normal), r, g, b (color), q (quality), rad, vi, and all custom *vertex attributes* already defined by user.
- ***Conditional Face Selection*** Boolean function using muparser lib to perform faces selection over current mesh.
  It's possible to use parenthesis, per-vertex variables and boolean operator:
  (,),**and**,**or**,**<>**,**=**
  It's possible to use per-face variables like attributes associated to the three vertex of every face.
  x0,y0,z0 for **first vertex**; x1,y1,z1 for second vertex; x2,y2,z2 for third vertex.

and also **nx0,ny0,nz0** nx1,ny1,nz1 etc. for **normals** and **r0,g0,b0** for **color**,**q0,q1,q2** for **quality**.

- ***Geometric Function*** Geometric function using muparser lib to generate new CoordYou can change x,y,z for every vertex according to the function specified.It's possible to use the following per-vertex variables in the expression:x, y, z, nx, ny, nz (normal), r, g, b (color), q (quality), rad, vi, and all custom *vertex attributes* already defined by user.
- ***Per-Face Color Function*** Color function using muparser lib to generate new RGB color for every face. Insert three function each one for red, green and blue channel respectively. It's possible to use per-face variables like attributes associated to the three vertex of every face. **x0,y0,z0** for **first vertex**; x1,y1,z1 for second vertex; x2,y2,z2 for third vertex and also **nx0,ny0,nz0** nx1,ny1,nz1 etc. for **normals** and **r0,g0,b0** for **color**,**q0,q1,q2** for **quality**.
- ***Per-Vertex Color Function*** Color function using muparser lib to generate new RGB color for every vertex. Insert three function each one for red, green and blue channel respectively. It's possible to use the following per-vertex variables in the expression: x, y, z, nx, ny, nz (normal), r, g, b (color), q (quality), rad, vi, and all custom *vertex attributes* already defined by user.
- ***Per-Vertex Quality Function*** Quality function using muparser to generate new Quality for every vertex. It's possible to use the following per-vertex variables in the expression: x, y, z, nx, ny, nz (normal), r, g, b (color), q (quality), rad, vi, and all custom *vertex attributes* already defined by user.
- ***Per-Face Quality Function*** Quality function using muparser to generate new Quality for every face. Insert three function each one for quality of the three vertex of a face. It's possible to use per-face variables like attributes associated to the three vertex of every face. **x0,y0,z0** for **first vertex**; x1,y1,z1 for second vertex; x2,y2,z2 for third vertex. and also **nx0,ny0,nz0** nx1,ny1,nz1 etc. for **normals** and **r0,g0,b0** for **color**,**q0,q1,q2** for **quality**.
- ***Define New Per-Vertex Attribute*** Add a new Per-Vertex scalar attribute to current mesh and fill it with the defined function. The name specified below can be used in other filter functionIt's possible to use the following per-vertex variables in the expression: x, y, z, nx, ny, nz (normal), r, g, b (color), q (quality), rad, vi, and all custom *vertex attributes* already defined by user.
- ***Define New Per-Face Attribute*** Add a new Per-Face attribute to current mesh. You can specify custom name and a function to generate attribute's value. It's possible to use per-face variables in the expression: **x0,y0,z0** for **first vertex**; x1,y1,z1 for second vertex; x2,y2,z2 for third vertex. and also **nx0,ny0,nz0** nx1,ny1,nz1 etc. for **normals** and **r0,g0,b0** for **color**,**q0,q1,q2** for **quality**. name specified below can be used in other filter function
- ***Grid Generator*** Generate a new 2D Grid mesh with number of vertices on X and Y axis specified by user with absolute length/height. It's possible to center Grid on origin.
- ***Implicit Surface*** Generate a new mesh that corresponds to the 0 valued isosurface defined by the scalar field generated by the given expression

- ***Refine User-Defined*** Refine current mesh with user defined parameters. Specify a Boolean Function needed to select which edges will be cut for refinement purpose. Each edge is identified with first and second vertex. Arguments accepted are first and second vertex attributes: **x0,y0,z0 x1,y1,z1** for coord **nx0,ny0,nz0 nx1,ny1,nz1** for normal **r0,g0,b0 r1,g1,b1** for color **q0 q1** for quality. Coords for new vertex on edge are generated with function x,y and z You can use **x0,y0,z0** and **x1,y1,z1**
- ***Loop Subdivision Surfaces*** Apply Loop's Subdivision Surface algorithm. It is an approximate method which subdivide each triangle in four faces. It works for every triangle and has rules for extraordinary vertices
- ***Butterfly Subdivision Surfaces*** Apply Butterfly Subdivision Surface algorithm. It is an interpolated method, defined on arbitrary triangular meshes. The scheme is known to be C1 but not C2 on regular meshes
- ***Remove Unreferenced Vertex*** Check for every vertex on the mesh if it is referenced by a face and removes it
- ***Remove Duplicated Vertex*** Check for every vertex on the mesh if there are two vertices with same coordinates and removes it
- ***Remove Zero Area Faces*** Removes null faces (the one with area equal to zero)
- ***Remove Faces with edges longer than...*** Remove from the mesh all triangles whose have an edge with lenght greater or equal than a threshold
- ***Clustering decimation*** Collapse vertices by creating a three dimensional grid enveloping the mesh and discretizes them based on the cells of this grid
- ***Quadric Edge Collapse Decimation*** Simplify a mesh using a Quadric based Edge Collapse Strategy, better than clustering but slower
- ***Quadric Edge Collapse Decimation (with texture)*** Simplify a textured mesh using a Quadric based Edge Collapse Strategy, better than clustering but slower
- ***Midpoint Subdivision Surfaces*** Apply a plain subdivision scheme where every edge is split on its midpoint
- ***Re-Orient all faces coherently*** Re-orient in a consistent way all the faces of the mesh
- ***Invert Faces Orientation*** Invert faces orientation, flip the normal of the mesh
- ***Remove Non Manifold Faces*** Remove non 2-manifold edges by removing some of the faces incident on non manifold edges
- ***Remove Non Manifold Vertices*** Remove non 2-manifold vertices, that vertices where the number of faces that can be reached using only face-face connectivity is different from the number of faces actually incident on that vertex.Typical example think to two isolated triangles connected by a single vertex building a *hourglass* shape.
- ***Compute normals for point sets*** Compute the normals of the vertices of a mesh without exploiting the triangle connectivity, useful for dataset with no faces
- ***Compute curvature principal directions*** Compute the principal directions of curvature with several algorithms

- ***Close Holes*** Close holes smaller than a given threshold
- ***Freeze Current Matrix*** Freeze the current transformation matrix into the coords of the vertices of the mesh
- ***Apply Transform*** Apply transformation, you can rotate, translate or scale the mesh
- ***Geometric Cylindrical Unwrapping*** Unwrap the geometry of current mesh along a clylindrical equatorial projection. The cylindrical projection axis is centered on the origin and directed along the vertical **Y** axis.
- ***MLS projection (RIMLS)*** Project a mesh (or a point set) onto the MLS surface defined by itself or another point set. This is the Robust Implicit MLS (RIMLS) variant which is an extension of Implicit MLS preserving sharp features using non linear regression. For more details see: Oztireli, Guennebaud and Gross, 'Feature Preserving Point Set Surfaces based on Non-Linear Kernel Regression' Eurographics 2009.
- ***MLS projection (APSS)*** Project a mesh (or a point set) onto the MLS surface defined by itself or another point set. This is the *algebraic point set surfaces* (APSS) variant which is based on the local fitting of algebraic spheres. It requires points equipped with oriented normals. For all the details about APSS see: Guennebaud and Gross, 'Algebraic Point Set Surfaces', Siggraph 2007, andGuennebaud et al., 'Dynamic Sampling and Rendering of APSS', Eurographics 2008
- ***Marching Cubes (RIMLS)*** Extract the iso-surface (as a mesh) of a MLS surface defined by the current point set (or mesh)using the marching cubes algorithm. The coarse extraction is followed by an accurate projectionstep onto the MLS, and an extra zero removal procedure. This is the Robust Implicit MLS (RIMLS) variant which is an extension of Implicit MLS preserving sharp features using non linear regression. For more details see: Oztireli, Guennebaud and Gross, 'Feature Preserving Point Set Surfaces based on Non-Linear Kernel Regression' Eurographics 2009.
- ***Marching Cubes (APSS)*** Extract the iso-surface (as a mesh) of a MLS surface defined by the current point set (or mesh) using the marching cubes algorithm. The coarse extraction is followed by an accurate projection step onto the MLS, and an extra zero removal procedure. This is the *algebraic point set surfaces* (APSS) variant which is based on the local fitting of algebraic spheres. It requires points equipped with oriented normals. For all the details about APSS see: Guennebaud and Gross, 'Algebraic Point Set Surfaces', Siggraph 2007, and Guennebaud et al., 'Dynamic Sampling and Rendering of APSS', Eurographics 2008
- ***Colorize curvature (RIMLS)*** Colorize the vertices of a mesh or point set using the curfvature of the underlying surface. This is the Robust Implicit MLS (RIMLS) variant which is an extension of Implicit MLS preserving sharp features using non linear regression. For more details see: Oztireli, Guennebaud and Gross, 'Feature Preserving Point Set Surfaces based on Non-Linear Kernel Regression' Eurographics 2009.
- ***Colorize curvature (APSS)*** Colorize the vertices of a mesh or point set using the curvature of the underlying surface. This is the *algebraic point set surfaces* (APSS) variant which is based on the local fitting of algebraic

spheres. It requires points equipped with oriented normals. For all the details about APSS see: Guennebaud and Gross, 'Algebraic Point Set Surfaces', Siggraph 2007, and Guennebaud et al., 'Dynamic Sampling and Rendering of APSS', Eurographics 2008

- ***Estimate radius from density*** Estimate the local point spacing (aka radius) around each vertex using a basic estimate of the local density.
- ***Small component selection*** Select the small disconnected components of a mesh.
- ***Poisson Reconstruction*** Use the points and normal to build a surface using the Poisson Surface reconstruction approach.
- ***Quality Mapper applier*** The filter maps quality levels into colors using a colorband built from a transfer function (may be loaded from an external file) and colorizes the mesh vertexes. The minimum, medium and maximum quality values can be set by user to obtain a custom quality range for mapping
- ***Mesh Element Subsampling*** Create a new layer populated with a point sampling of the current mesh, At most one sample for each element of the mesh is created. Samples are taking in a uniform way, one for each element (vertex/edge/face); all the elements have the same probabilty of being choosen.
- ***Montecarlo Sampling*** Create a new layer populated with a point sampling of the current mesh; samples are generated in a randomly uniform way, or with a distribution biased by the per-vertex quality values of the mesh.
- ***Stratified Triangle Sampling*** Create a new layer populated with a point sampling of the current mesh; to generate multiple samples inside a triangle each triangle is subdivided according to various *stratified* strategies. Distribution is often biased by triangle shape.
- ***Poisson-disk Sampling*** Create a new layer populated with a point sampling of the current mesh; samples are generated according to a Poisson-disk distribution
- ***Variable density Disk Sampling*** Create a new layer populated with a point sampling of the current mesh; samples are generated according to a Poisson-disk distribution
- ***Hausdorff Distance*** Compute the Hausdorff Distance between two meshes, sampling one of the two and finding foreach sample the closest point over the other mesh.
- ***Texel Sampling*** Create a new layer with a point sampling of the current mesh, a sample for each texel of the mesh is generated
- ***Vertex Attribute Transfer*** Transfer the choosen per-vertex attributes from one mesh to another. Useful to transfer attributes to different representations of a same object. For each vertex of the target mesh the closest point (not vertex!) on the source mesh is computed, and the requested interpolated attributes from that source point are copied into the target vertex. The algorithm assumes that the two meshes are reasonably similar and aligned.
- ***Uniform Mesh Resampling*** Create a new mesh that is a resampled version of the current one. The resampling is done by building a uniform volumetric representation where each voxel contains the signed distance

from the original surface. The resampled surface is reconstructed using the **marching cube** algorithm over this volume.

- ***Voronoi Vertex Clustering*** Apply a clustering algorithm that builds voronoi cells over the mesh starting from random points,collapse each voronoi cell to a single vertex, and construct the triangulation according to the clusters adjacency relations. Very similar to the technique described in **'Approximated Centroidal Voronoi Diagrams for Uniform Polygonal Mesh Coarsening'** - Valette Chassery - Eurographics 2004
- ***Voronoi Vertex Coloring*** Given a Mesh **M** and a Pointset **P**, The filter project each vertex of P over M and color M according to the geodesic distance from these projected points. Projection and coloring are done on a per vertex basis.
- ***Disk Vertex Coloring*** Given a Mesh **M** and a Pointset **P**, The filter project each vertex of P over M and color M according to the geodesic distance from these projected points. Projection and coloring are done on a per vertex basis.
- ***Regular Recursive Sampling*** The bbox is recrusively partitioned in a octree style, center of bbox are considered, when the center is nearer to the surface than a given thr it is projected on it. It works also for building ofsetted samples.
- ***Select All*** Select all the faces of the current mesh
- ***Select None*** Clear the current set of selected faces
- ***Delete Selected Faces*** Delete the current set of selected faces, vertices that remains unreferenced are not deleted.
- ***Delete Selected Faces and Vertices*** Delete the current set of selected faces and all the vertices surrounded by that faces.
- ***Erode Selection*** Erode (reduce) the current set of selected faces
- ***Dilate Selection*** Dilate (expand) the current set of selected faces
- ***Select Border Faces*** Select all the faces on the boundary
- ***Invert Selection*** Invert the current set of selected faces
- ***Select by Vertex Quality*** Select all the faces with all the vertexes within the specified quality range
- ***Select Face by Vertex Color*** Select part of the mesh based on its color.
- ***Move selection on another layer*** Selected faces are moved (or duplicated) in a new layer
- ***Duplicate current layer*** Create a new layer containing the same model as the current one
- ***Planar flipping optimization*** Mesh optimization by edge flipping, to improve local triangle quality
- ***Curvature flipping optimization*** Mesh optimization by edge flipping, to improve local mesh curvature
- ***Laplacian smooth (surface preserve)*** Laplacian smooth without surface modification: move each vertex in the average position of neighbors vertices, only if the new position still (almost) lies on original surface
- ***Cut mesh along crease edges*** Cut the mesh along crease edges, duplicating the vertices as necessary. Crease edges are defined according to the variation of normal of the adjacent faces

- ***Laplacian Smooth*** Laplacian smooth of the mesh: for each vertex it calculates the average position with nearest vertex
- ***HC Laplacian Smooth*** HC Laplacian Smoothing, extended version of Laplacian Smoothing, based on the paper of Vollmer, Mencl, and Muller
- ***ScaleDependent Laplacian Smooth*** Scale Dependent Laplacian Smoothing, extended version of Laplacian Smoothing, based on the Fujiwara extended umbrella operator
- ***TwoStep Smooth*** Two Step Smoothing, a feature preserving/enhancing fairing filter. It is based on a Normal Smoothing step where similar normals are averaged together and a step where the vertexes are fitted on the new normals
- ***Taubin Smooth*** The $lambda-mu$ taubin smoothing, it make two steps of smoothing, forth and back, for each iteration
- ***Depth Smooth*** A laplacian smooth that is constrained to move vertices only along the view direction.
- ***Directional Geom. Preserv.*** Store and Blend the current geometry with the result of another previous smoothing processing step. It is useful to limit the influence of any smoothing algorithm along the viewing direction. This is import to cope with the biased distribution of the error in many scanning devices. TOF scanner usually have very good **x,y** accuracy but suffer of great depth errors.
- ***Smooth vertex quality*** Laplacian smooth of the quality values.
- ***Smooth Face Normals*** Smooth Face Normals without touching the position of the vertices.
- ***UnSharp Mask Normals*** Unsharp mask filtering of the normals, putting in more evidence normal variations
- ***UnSharp Mask Geometry*** Unsharp mask filtering of geometric shape, putting in more evidence ridges and valleys variations
- ***UnSharp Mask Quality*** Unsharp mask filtering of the quality field
- ***UnSharp Mask Color*** Unsharp mask filtering of the color, putting in more evidence color edge variations
- ***Recompute Vertex Normals*** Recompute vertex normals as an area weighted average of normals of the incident faces
- ***Recompute Weighted Vertex Normals*** Recompute vertex normals as a weighted sum of normals of the incident faces. Weights are defined according to the paper *Weights for Computing Vertex Normals from Facet Normals*, Nelson max, JGT 1999
- ***Recompute Angle Weighted Vertex Normals*** Recompute vertex normals as an angle weighted sum of normals of the incident faces according to the paper *Computing Vertex Normals from Polygonal Facet*, G Thurmer, CA Wuthrich, JGT 1998
- ***Recompute Face Normals*** Recompute face normals as the normal of the plane of the face
- ***Normalize Face Normals*** Normalize Face Normal Lenghts
- ***Normalize Vertex Normals*** Normalize Vertex Normal Lenghts
- ***Vertex Linear Morphing*** Morph current mesh towards a target with the same number of vertices. The filter assumes that the two meshes have also the same vertex ordering.

- ***Remove border faces*** Remove all the faces that has at least one border vertex.
- ***Noisy Isosurface*** Create a isosurface perturbed by a noisy isosurface.
- ***Colorize by border distance*** Store in the quality field the geodesic distance from borders and color the mesh accordingly.
- ***Random vertex displacement*** Move the vertices of the mesh of a random quantity.
- ***Flatten visible layers*** Flatten all or only the visible layers into a single new mesh. Transformations are preserved. Existing layers can be optionally deleted
- ***Vertex Color Noise*** Randomly add a small amount of a random base color to the mesh

Simplification

Simplification filters are filters that get in input a mesh and build another mesh composed by a smaller number of triangles. There are three simplification filters inside MeshLab

- **Clustering**. Based on a simple vertex clustering approach. A uniform grid is used to collapse all the vertices falling in the same grid cell onto a single vertex. It is a very fast algorithm but not very accurate, a bound of the introduced error is half of the diagonal of each grid cell. Topological inconsistencies are quite probable that will be created (*non manifold edges*). This is a typical drawback of any clustering-based simplification algorithms. In our implementation the only parameter that can be specified by the user is the length of the side of the cubic grid cell. This length can be introduced either as an absolute value in the space of the mesh itself, or as a percentage of the diagonal of the bounding box of the object. With this kind of approach you cannot know in advance the final size of the mesh, but only knowing what could be at the end the final average edge length (e.g. approx. the same of the grid cell side).

- **Quadric edge collapse**. A variant of the well known edge collapse algorithm based on quadric error metric proposed by Michael Garland and Paul Heckbert . The user can control the quality of the simplification through some parameters:
  - Quality threshold for penalizing edge collapses that creates bad shaped faces.The value is in the range [0..1] (0 a degenerate triangle 1 a equilateral one); the value of this parameter must be interpreted as follows:
    - 0 accept any kind of face (no penalties),
    - *k* penalize faces with quality than *k*, proportionally to their shape quality.
  - Normal checking for penalizing edge collapses that flip the normal of the original surfaces (it can help to avoid some nasty artifacts that can arise in very flat areas).

- **Quadric edge collapse with texture coordinates preservation**

Point Picking Tool

Description
This tool can basically be described as a mesh labeling tool. It allows you to pick and name points on a mesh and save the output to xml.

The tool has two main modes, normal and template mode. Template mode is described below. There are two main modes of interaction with the points, **Pick Point**, **Move Point**, and **Select Point** mode. The mode of interaction is indicated by the radio at the top. In pick mode you add new points to the mesh by right clicking on the mesh. In move mode you can move a point by right clicking near the point and dragging it to a different location. In select mode you can right click a point on the mesh and it will be highlighted in the pick points edit tool menu.
Here are some notes about the **active** checkbox. The immediate affect of checking the active checkbox is to hide/show the picked point on the mesh. This feature is meant to allow you to use a template for which some scan data will have one of the template points missing. In this case you can unflag this point, labeling it in-active. When points are added you have a green cone, representing the normal, with a label next to it. You can also choose at the bottom where it says **Normal Options** to have just a dot, which is the point, a label for that point, or a dot, label, and a green line which represents the normal of the face the point is on.

Template Mode
Summary: Templates are use to facilitate the labeling of many meshes with the same labels. If you have a template loaded you are in Template Mode. The name of the template will be written in blue letters next to the place at the bottom where it says "Template Name:". "No Template Loaded" will be printed if no template is loaded. In Template mode you will only be able to pick and move the points that are already defined as part of the template. Use the **Add Point** and **Remove Point** buttons in the Template Controls area to modify a template. You can also start from scratch, pick some points, then save them as a template (saving the names so you can use them for other meshes).

Saving/Loading Points
Points are saved to **a xml file** by using the **save button** at the top. Points can be loaded from a file by pressing the load button. Alternately points are **automatically loaded from a file** if you had previously saved them using the filename recommended by the tool. Points are **saved to PerMeshAttributes** automatically when you close the edit plugin and loaded back from PerMeshAttribute when the plugin is loaded. This takes precedent over loading points from a file.

Other Features
1. Rename a point by slowly double clicking on it
2. Undo: will undo only that last move

Use With "Align Mesh using Picked Points" filter
In order to use the **Align Mesh using Picked Points** filter you need to have either saved points:

1. to PerMeshAttributes or
2. to a file with the name suggested by the point picking plugin.

Note: that points are only needed if "Use Markers for Alignment" is checked.
Note: that if "Scale the mesh" is checked then you need points and to have checked "Use Markers for Alignment".
Note: the transform calculated is stored in per mesh data with key AlignTools::getKey()

Notes for Developers who want to use picked points in their plugin

You can look at how the filter FP_ALIGN_WITH_PICKED_POINTS in the cleanfilter plugin uses points picked by the edit_pickpoints plugin. Note, most code is here, cleanfilter/cleanfilter.cpp but the important bit is here:

```
MeshModel *stuckModel = par.getMesh(StuckMesh); PickedPoints *stuckPickedPoints = 0; ... //first try to get points from memory if(vcg::tri::HasPerMeshAttribute(stuckModel->cm, PickedPoints::Key) ) {
        CMeshO::PerMeshAttributeHandle<PickedPoints*> ppHandle = vcg::tri::Allocator<CMeshO>::GetPerMeshAttribute<PickedPoints*>(stuckModel->cm, PickedPoints::Key); stuckPickedPoints = ppHandle();
```

What you get is a PickedPoints object. To see what it does look here /meshlabplugins/edit_pickpoints/pickedPoints.h(cpp).
Basically what you need to know is that this class hold a std::vector<PickedPoint*> where a PickedPoint is a simple object that has two members:

1. name of point: QString name;
2. point: vcg::Point3f point;

Other information

- source code in edit_pickpoints folder

Morpher Tool

Description
This is a fairly simple plugin that allows you to morph between any two meshes that have the same number of vertexes. The morph is calculated and displayed in real time as you slide the slider.

Details

- The morph is a simple linear morph between the corresponding points. Example: A morph of 50 calculates the point 50% between the first point of mesh A and the first point of mesh B, then goes on to the second point, etc
- NOTE: does compact the mesh removeing deleted elements using Allocator<MyMesh>::CompactVertexVector(m);and Allocator<MyMesh>::CompactFaceVector(m);

Other information

- source code in filter_morph folder

MLS Surfaces

Moving least squares (MLS) filters provide suite of tools built upon various meshless surface definitions. The set of supported MLS surfaces variants include:

- Adamson and Alexa's point set surfaces based on normal averaging (special case of APSS),
- Guennebaud and Gross's algebraic point set surfaces (APSS) which are based of sphere fitting,
- Kolluri's implicit MLS variant of Shen et al. approach.

Note that all these definitions require vertices equipped with oriented normals. If no local radius is provided, then they are automatically computed using the Point Set/Radius estimation filter. All MLS filters take as input a point set (or the vertices of a mesh), implicitly define the underlying surface, and then perform various operations such as:

- **Meshing** of the surface using a marching cube algorithm.
- **Smoothing** by mean of a projection onto the underlying MLS surface. It is possible to project/smooth the input vertices themself, or any other set of points of meshing assuming they are close enough to the target surface (currently). If the geometry to project is a mesh, then it is possible to refine it during the projection according to some local error metrics (here the angle between two adjacent faces).
- **Curvature plots**: colorizes the vertices of a point set (or mesh) according to the curvature of the underlying MLS surface. It provides exact analytics curvature computations for the Gauss, mean and principal curvatures.

Filters highly related to point set surfaces includes:

- **Point Set/Radius estimation** which computes the local point spacing of each vertices of a point set. This filter use a basic estimate of the local density using the k nearest neighbors and assuming a locally uniform and flat sampling.
- **Normals/Compute from point set** which can be used to compute oriented normals of a raw point cloud.
- **Select/Small components** which can be used to select (and then remove) all the extra zero components which can appears after a marching cube reconstruction of the MLS surface.

Qhull Filter

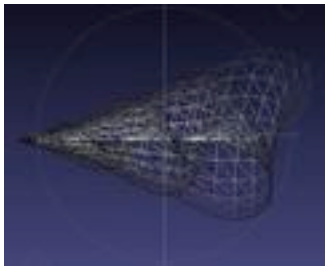
General Description

The Qhull Filter plugin is a collection of filters used for computing the convex hull, Delaunay triangulation, Voronoi Filtering and Alpha Shapes over the mesh. It also includes a filter for the selection of the visible points from a given viewpoint. Every single filters uses the Qhull library for mathematical operations.

Function Filter List

Convex Hull

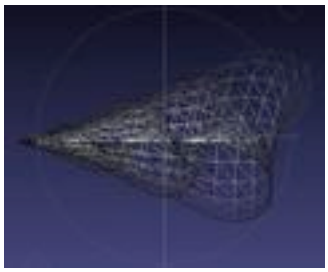
This filter computes the *convex hull* of the current mesh. It is contained in the Remeshing filter class. The convex hull of a set of points is the boundary of the minimal convex set containing the given non-empty finite set of points. The filter takes the mesh vertices and builds a new mesh that contains the convex hull. Here it is the Qhull library documentation of the program that calculates the convex hull of a set of points. The result is 2-manifold by definition, so it is possible to re-orient all faces coherently.

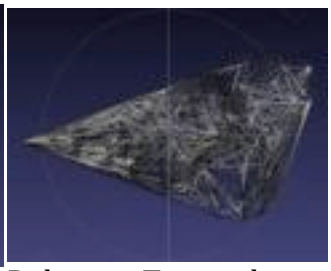

Original image


Convex Hull

Delaunay Triangulation

This filter computes the *Delaunay triangulation* of the current mesh. It is contained in the Remeshing filter class. The Delaunay triangulation DT(S) of a set of points S in d-dimensional spaces is a triangulation of the convex hull such that no point in S is inside the circum-sphere of any simplex in DT(S). The filter takes the mesh vertices and builds a new mesh that contains the Delaunay triangulation. Here it is the Qhull library documentation of the program that calculates the Delaunay triangulation of a set of points S.


Original image


Delaunay Triangulation

Voronoi Filtering

The filter implements a *Voronoi filtering* on the current mesh. It is contained in the Remeshing filter class. The algorithm computes a piecewise-linear approximation of a smooth surface from a finite set of sample points that defines a "well-sampled" curve. It uses a subset of the Voronoi vertices to remove triangles from the Delaunay triangulation. The algorithm picks only two Voronoi vertices, called *poles*, per sample point: the farthest vertices of the point's cell on each side of the surface.

Here it is the main steps proposed by *Amenta and Bern* (1998):

1. Compute the Voronoi diagram of the sample points S.
2. For each sample point s do:
    1. If s does not lie on the convex hull of S, let p+ be the vertex of Vor(s) farthest from s.
    2. If s does lie on the convex hull, let p+ be a point at "infinite distance" outside the convex hull with the direction of sp+ equal to the average of the outward normals of hull faces meeting at s.
    3. Pick the vertex p- of Vor(s) farthest from s with negative projection onto sp+.
3. Let P denotes all poles p+ and p-,except those p+'s at infinitive distance. Compute the Delaunay triangulation of S U P.
4. (Voronoi filtering) Keep only those triangles in which all three vertices are sample points.

The filter allows the user to choose a *threshold*, a factor that defines a range (0-threshold*bbox.Diag()) used to discard the Voronoi vertices too far from the origin. They can cause problems to the qhull library. Growing values of *threshold* will add more Voronoi vertices for a better surface reconstruction.

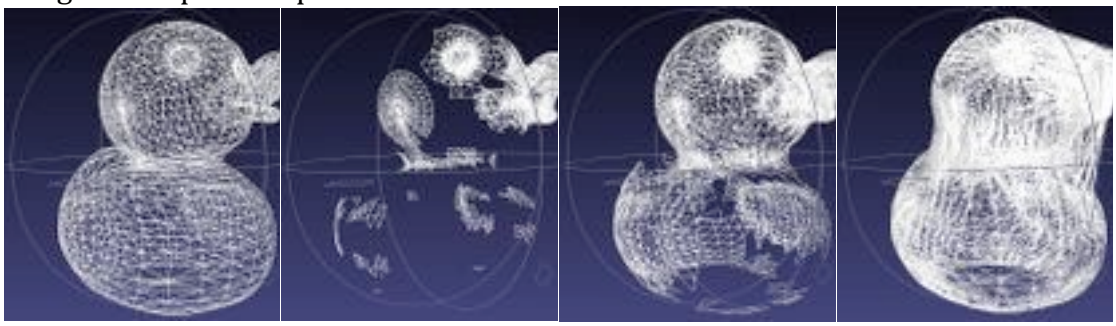
Alpha Shapes

The filter computes the *Alpha Shape* (Edelsbrunner and P.Mucke 1994) of the current mesh. It is contained in the Remeshing filter class. From a given finite point set in the space it computes the *shape* of the set. The alpha shape is the boundary of the alpha complex, that is a subcomplex of the Delaunay triangulation of the given point set S. For a given value of 'alpha', the alpha complex includes all the simplices in the Delaunay triangulation which have an empty circumsphere with radius equal or smaller than 'alpha'. Here 'empty' means that the open shpere do not include any points of S.

The algorithm uses the general-position assumption by Edelsbrunner and P.Mucke that states that no smallest circumsphere of two or three points of S contains another point of S.

If a sphere contains three points of S then no two of them are antipodal, and if it contains four points then no three lie on a great-circle of the sphere. (*ACM Transition on Graphics, Vol. 13, no 1, January 1994*). 'alpha' controls the desired level of detail of the shape. Note that for 'alpha' = 0, both the alpha complex and the alpha shape consist just of the set S, and for sufficiently large 'alpha', the alpha complex is the Delaunay triangulation DT(S) of S, instead the alpha shape is the convex hull of S. The alpha value is computed as percentage of the diagonal of the box. User can choose to get the alpha complex or the alpha shape of the mesh. The filter inserts the minimum value of alpha (the circumradius of the triangle) in attribute Quality for each face, so it can be easier to define a filtration on the mesh (*the alpha shapes*).

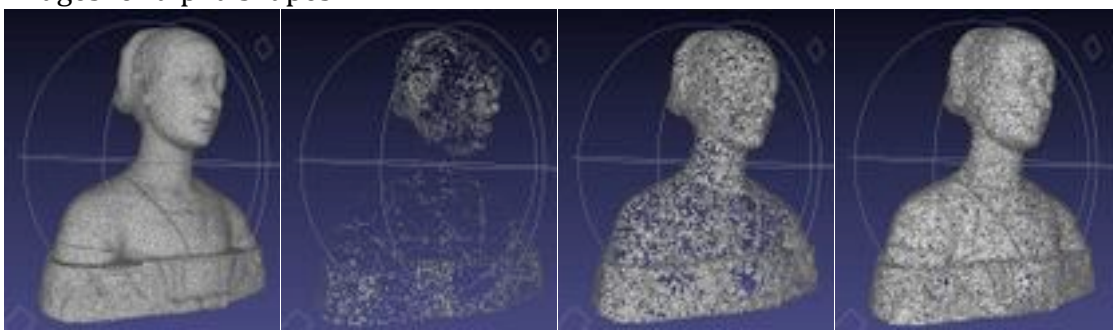
Images for alpha complexes



Original image Alpha=51.34 Alpha=85.57 Alpha=290.92

.

Images for alpha shapes



Original image Alpha=2 Alpha=3 Alpha=4

Select Visible Points

The filter selects the visible vertices of the current mesh, as viewed from a given viewpoint. It is contained in the Selection and Point Set filter class. The algorithm used (Katz, Tal and Basri 2007) determines visibility without reconstructing a surface or estimating normals. A point is considered visible if its transformed point lies on the convex hull of a transformed points cloud from the original given points. Here it is the equation used to perform the spherical flipping (Katz et al. 2005):

f(p)= p + 2*(R - ||p||)*p/||p||

It is important to choose a suitable value for R. As R increases, more points pass the threshold of the convex hull and hence are marked as visible. A large R is suitable for dense point clouds, and a small R for sparse clouds. The radius is computed as the distance between the viewpoint and its farthest point and then adjusted with a parameter (*threshold*)inserted by the user. radius=distance between the center and its farthest point * 10^threshold. User can choose to use the Meshlab Camera as the viewpoint or insert his own one. User can also choose to show the partial convex hull of the transformed points (the viewpoint is not included) and a mesh connecting the visible points according to the connectivity of the flipped ones. It is also possible to re-orient all faces coherently.

Supported Formats

The following formats are supported by MeshLab

Import

- PLY, ascii and binary. Many variants supported. Triangle strip loading supported. ply Range map format still not supported
- STL, ascii and binary. Color not supported.
- OFF,
- OBJ,
- 3DS,
- COLLADA,
- PTX

Export

- PLY,
- STL,
- OFF,
- OBJ,
- 3DS,
- COLLADA,
- VRML,
- DXF
- U3D

Compiling devel

What you need

To compile MeshLab you need a C++ compiling environment, (we regularly compile meshlab under VisualStudio 2005, gcc and xcode) and the following libraries:

- Qt 4.4 (note that Qt 4.4 are *required*, Qt versions < 4.4 could not compile). Current version of MeshLab compiles well against Qt 4.4.3.
- A svn-client (for a list of existing svn-clients you could take a look here).
- the VCG libraries; http://vcg.sf.net, this library is not distributed by means of easy-to-download packages, but it is only accessible through anonymous svn (svn string: *svn co https://vcg.svn.sourceforge.net/svnroot/vcg vcg*). If you get the current version of the VCG library we suggest you to also get the current version of MeshLab. If you want to compile the distributed sources check out from the repository the VCG lib with the same date of the release of MeshLab.
- MeshLab's source code available via SourceForge svn server. For further details please refers to the following section

Getting the sources

For getting the latest version of MeshLab you have to anonymously access the MeshLab **svn** tree as explained in the sourceforge page http://sourceforge.net/svn/?group_id=149444; for the lazy ones, to get the right subset of the svn trunk in the **devel** folder you should issue the following svn command:

svn co https://meshlab.svn.sourceforge.net/svnroot/meshlab/trunk/meshlab meshlab svn co https://vcg.svn.sourceforge.net/svnroot/vcg/trunk/vcglib vcglib

External Dependencies

MeshLab and MeshLab's plugins invokes functions exported by external libraries. The code of all these libraries is included in the MeshLab's current development version downloadable via svn. You can find it in the folder ~/devel/meshlab/src/external.

- glew (http://glew.sourceforge.net/), strongly required. You just need the sources (.h and .c), (not the compiled libs), glew is statically compiled into meshlab, so no dynamic libraries are required.
- lib3ds-1.3.0 (http://lib3ds.sourceforge.net/) This one is needed only for the io_3ds plugin. Without this plugin the 3ds file formats will not be parsed.
- bzip2-1.0.3 this one is required only for the epoch format import plugins; if you do not have datasets produced with the arc3D web reconstruction service, you do not need this plugin.
- muParser 1.30 (http://muparser.sourceforge.net/) This library is needed for filter_func plugin.

In order to successfully compile a MeshLab's plugin with external dependency you have before to compile the referred library and put it into the folder ~/devel/meshlab/src/external/lib/YOUR_OS where YOUR_OS is a string related to your operating system and compiler according to QMAKESPEC definition (like for example *macx* or *win32-msvc2005*).
Into the folder ~/devel/meshlab/src/external a .pro file, external.pro, has been provided to easily compile all the external libraries. To do this you can either:

- use a Qt-friendly IDE, like Qt Creator, Visual Studio or Eclipse (the last two with the IDE's Qt integration provided by Trolltech)
- use command line.

qmake -recursive external.pro make

A note: when we say that a library is needed only for a specific plugin we mean that if you do not have that library you should not try to compile that specific plugin otherwise the compilation of that plugin could fail on that plugin and, depending on your compiling environment, the whole compilation could stop. To avoid the compilation of problematic plugins simply comments the relative line (adding a # at the beginning of the line) in the src/meshlabplugins/meshlabpluginsv12.pro.

Code tree structure

The meshlab sources and the vcg library must be at the same level and the vcg library root should be named 'vcglib'. The external compiled libraries should be placed in a dir named ~/devel/meshlab/external/lib/YOUR_OS (see the above section).
You should have setup your directories in a way that should be similar to the following ones (obviously you are not forced to call your base dir devel or to put it under your home dir):

~/devel/vcglib/vcg/space/ ~/devel/meshlab/src/meshlab/interfaces.h ~/devel/meshlab/external/lib/YOUR_OS
If you put things in dirs with different relations, you have to manually change the .pro. Please note that if you are a developer that will soon or later submit back its work, it is strongly deprecated that you use your own version of .pro files: use the standard setup.

Compiling
The compiling step depends on the compiling environment. Using GCC (both under linux and using the mingw gcc provided with the free Qt distribution) you should just type, from the *devel/meshlab/src* directory:
qmake -recursive meshlabv12.pro make
the *devel/meshlab/src* directory contains also the *meshlab_mini.pro* project that contains a much smaller set of the meshlab plugins. It is easier to cater for (no need for lib3ds and bzip) and compiles faster.
Under windows the suggested platform is the one formed by the open source version of Qt with the mingw gcc compiler that is kindly included in the open source Qt distribution available from TrollTech.
If you want to use Visual Studio, please *buy* the commercial version of Qt that offers a nice integration of the Qt tools into the Visual Studio IDE. In that case you simply have to import the top level pro (*~/devel/meshlab/src/meshlab.pro*) into VisualStudio.

Using Microsoft Visual Studio 8 express edition and QT
It is possible to use the Visual Studio 8 express edition (the one free from MS). Simple steps for the lazy ones:

- download and install Microsoft Visual Studio 8 express edition
- download from trolltech QT 4.4.3 , the zip with the sources not the precompiled bin for mingw.
- unzip it and rename the created folder to c:\Qt\4.4.3 (or something different if you already have installed the qt distribution with mingw.
- Now open the visual studio console (program files->visual studio 2008-> vs 2008 tools-> vs 2008 command promt). Go to the directory where you unzipped qt and type configure -debug-and-release. Wait a few mins (it compiles qmake and runs it, creating the makefiles for vs2008)
- type nmake. Wait a few hours (it recompiles the whole Qt).
- last step: prepare the environment; you have to add to the environment of vs8 command prompt:

set QTDIR=C:\Qtvs\4.4.3 set PATH=C:\Qtvs\4.4.3\bin set PATH=%PATH%;%SystemRoot%\System32 set QMAKESPEC=win32-msvc2008

- after that just start that enhanced vs8 command prompt, go in the appropriate dir, type

qmake -tp vc -recursive meshlabv12.pro

- and magically a solution that contains meshlab and all the plugins will appear.
- In some cases it could help also to configure visual C++, so it can see always Qt: (tools->options) and add: $(QTDIR)\include to the include directories and $(QTDIR)\lib to the lib directories, but it should not be required (and perhaps a bit dangerous if you upgrade qt).

Writing a Plugin

MeshLab has a reasonably modular architecture, most of its functionalities are implemented as dynamically loaded plugins. To extend meshlab with new functionalities you have to write new plugins. There is a fixed small set of possible plugins types for meshlab, the most important ones are:

- **filter plugins**. The most common class of plugins. These plugins implement straight mesh filtering operations, e.g. procedures that take in input a mesh and a set of parameters and modifies it according to some algorithm (smoothing, cleaning, simplification etc). Note that the parameter asking through a dialog is usually managed by the meshlab framework. These filters can be used also from meshlabserver the command line version of the meshlab.
- **i/o plugins**. These plugins implement file parsing mechanism. They can require parameters before and after the processing of the file, for example to decide if a mesh has to be saved in binary format or not, or to load only a subset of the original data, or to perform some kind of standard postprocessing on the input mesh (like for example stl vertex unification).
- **edit plugins**. These plugins implement functionalities that require non-trivial user input and direct mesh interaction, like painting, selection, measuring and that could require a complex custom dialog.
    - NOTE: The edit plugins cannot be used from meshlabserver.
    - MeshEditInterface Implementation Guide

You can find very simple examples of the above described plugins in the *meshlab/src/sampleplugins/* directory.
Obviously following the coding recommendation is always warmly suggested.
The new plugins should be named according to the following naming strategy:

- edit plugins should be named *edit_something*
- filter plugins should be named *filter_something*
- io plugins should be named *io_something*

The name of the plugin and of the .pro file should be the same of the directory name.
Obviously, once you have succeed to set up the framework for your plugin, you have to actually make something on your mesh. For an introduction on how access the elements of the mesh and how to use/modify them look at the pages that explains how accessing the mesh and what algorithms are present in the vcg library.

Mesh Data structure

Two cents on the basic types inside MeshLab, whose declaration are in meshmodel.h:

- the basic document type is a **MeshDocument** that is basically a list of **MeshModel**.
- each **MeshModel** corresponds to one layer and contains a vcg mesh of type **CMeshO**

- **CMeshO** is the vcg mesh type (a vcg::trimesh) that you should provide to the various vcg Algorithms and from which you should get FaceIterators, VertexIterators and others types...

So for example when you open a file, a new MeshDocument is created, with a single layer that contains the loaded mesh. Some filters can create or destroy layers, for example the sampling filters usually create a new layer containing the sampled point clouds. Filters can work or on a single layer or on all the layers.
It is very important that filters keep coherent and updated the set of elements that are currently active for a given mesh, like for example colors, curvature, adjacency etc.

Writing your first filter
First suggestion, start with writing a plugin of the filter class. They are the easiest one, no interaction has to be managed, no dialog has to be designed, you have only to do the following steps:

- think to your plugin in terms of something taking a mesh and some parameters and working on it.
- define the requirements for your mesh (do you need any kind of topology? colors? etc)
- define the parameters using the standard method of the filter class
- define the apply method that uses the provided parameters on a given mesh.

Look at the sample in the sampleplugins directory, at the 'samplefilter' plugin and try to modify it a bit. Remember to change the name of the class you are deriving for your plugin when you start with your own plugin.

SVN
For committing back your code please follow CLOSELY the suggestions in the Writing Code page. Obviously for doing that you need to be a registered MeshLab Developer on the sourceforge project.

What is VCG Lib ?
The VCG Lib could be defined as a library for managing triangle meshes, but that would be unfair. VCG Lib is a library for managing *0,1,2* and *3*-simplicial complexes, which means Point Sets, Edge Mesh, Triangle Mesh and Tetrahedral Meshes. To be completely honest, it is starting to support polygon meshes as well (stay tuned).
It just happens that the research interests of the developers leaded to develop mesh processing algorithms mostly for triangle meshes, but the kernel is there waiting... However, to avoid annoying the reader with 0-simplices, maximal simplices and so on, in this tutorial we will always refer to the case of triangle meshes, extending the explanation to the other cases only where appropriate.

please note: this is not a reference manual, but a short tutorial to guide you through the few basic concepts of VCG Lib. The idea is that after reading this tutorial and trying the few examples in apps/samples/ referred in the following, you will have all it takes to write your own code.

Installation and folder structure

Getting VCG Lib

VCG Lib uses a SVN repository. To get the right subset of the svn trunk in the devel folder you should issue the following svn command:
svn co https://vcg.svn.sourceforge.net/svnroot/vcg/trunk/vcglib vcglib
Windows users with tortoisesvn (http://www.tortoisesvn.net) installed should 1. create a folder named **vcglib**. 2. right-click in the newly created vcglib folder and choose SVN Checkout 3. put in the first parameter: https://vcg.svn.sourceforge.net/svnroot/vcg/trunk/vcglib 4. press ok and wait, it should start to check out the last revision of the whole tree.

Folder Structure

Main article: Folder structure

VCG Lib is mostly made of header files (and its core part it's *only* header files). Just download the tarball from here and uncompress it in a folder, e.g. named vcg, inside you compiler "include" directory. Afterwards, you will include the file you need. Inside vcg folder you will find 4 sub-folders:

- **vcg**: this is the core of the library, where all the algorithms and data structures are defined. This part is pure, quite heavily templated, C++ code with STL support for common data structures and algorithms. You will not find a single include from here to anything else than standard libraries. Note that core part is made of header files (.h files) only.
- **wrap**: here you will find the wrapping of VCG concepts towards specific needs/contexts/libraries. For example you will find all the code to import/export meshes from the hard disk in many formats, or the routines for rendering triangle meshes with OpenGL, supports for common GUI tools like a trackball, and so on..
- **apps**: this folder contains the command line applications developed with the VCG Lib. Many (much more) examples can be found in MeshLab. The apps/simple directory contains a sub-collection of very basic apps. A good starting point for beginners!
- **docs**: documentation lives here (including this tutorial)

Basic Concepts

How to define a mesh type

The VCG Lib encodes a mesh as a set of vertices and triangles (i.e. triangles for triangle meshes, tetrahedra for tetrahedral meshes). The following line will be a part of the definition of a VCG type of mesh:
class MyMesh : public vcg::TriMesh< std::vector<MyVertex>, std::vector<MyFace> > {}
vcg::TriMesh is the base type for a triangle mesh and it is templated on:

- the type of STL *container* containing the vertices
  - which in turn is templated on your **vertex type**
- the type of STL *container* containing the faces
  - which in turn is templated on your **face type**

The face and the vertex type are the crucial bits to understand in order to be able to take the best from VCG Lib. A vertex, an edge, a face and a tetrahedron are just an user defined (possibly empty) collection of attribute. For example you will probably expect MyVertex to contain the (x,y,z) position of the vertex, but what about the surface normal at the vertex?.. and the color? VCG Lib gives you a pretty elegant way to define whichever attributes you want to store in each vertex, face, or edge.

For example, the following example shows three valid definitions of MyVertex :
#include <vcg/simplex/vertex/base.h> #include <vcg/simplex/vertex/component.h> class MyFace; class MyEdge; class MyVertex0 : public vcg::VertexSimp2<MyVertex0, MyEdge, MyFace, vcg::vertex::Coord3d, vcg::vertex::Normal3f> {}; class MyVertex1 : public vcg::VertexSimp2<MyVertex1, MyEdge, MyFace, vcg::vertex::Coord3d, vcg::vertex::Normal3f, vcg::vertex::Color4b> {}; class MyVertex2 : public vcg::VertexSimp2<MyVertex2, MyEdge,MyFace> {};
vcg::VertexSimp2 is the VCG base class for a vertex belonging to a 2-simplex. If we were to define a tetraedral mesh, for example, we would have used vcg::VertexSimp3. The first 3 templates of vcg::VertexSimp2 must specify the type of all the simplicies involved in the mesh in ascending order: the vertex type itself, the type of edge and the type of triangle (and the type of tetrahedron for the tetrahedral meshes). It can be annoying when you see it but it is useful that every entity involved knows the type of the others and this is the way VCG Lib does it. As you can see the three definitions of MyVertex differ for the remaining template parameters. These specify which attributes will be stored with the vertex type: MyVertex0 is a type storing coordinates as a triple of doubles and normal as a triple of floats, MyVertex1 also store a color value specified as 4 bytes and MyVertex2 does not store any attribute, is an empty class. vcg::Coord3d, vcg::Normal3f, vcg::Color4b and many others are implemented in VCG, their complete list can be found here. You can place any combination of them as a template parameters of your vertex (your simplex) type.

Now we have all it takes for a working definition of MyMesh type:
/* apps/sample/trimesh_base/trimesh_definition.h */ #include <vector> #include <vcg/simplex/vertex/base.h> #include <vcg/simplex/vertex/component.h> #include <vcg/simplex/face/base.h> #include <vcg/simplex/face/component.h> #include <vcg/complex/trimesh/base.h> class MyEdge; class MyFace; class MyVertex : public vcg::VertexSimp2<MyVertex, MyEdge, MyFace, vcg::vertex::Coord3d, vcg::vertex::Normal3f> {}; class MyFace : public vcg::FaceSimp2<MyVertex, MyEdge, MyFace, vcg::face::VertexRef> {}; class MyMesh : public vcg::tri::TriMesh< std::vector<MyVertex>, std::vector<MyFace> > {}; int main() { MyMesh m; return 0; }
One more comment: vcg::VertexRef is an attribute that stores 3 pointers to the type of vertex, so implementing the Idexed Data structure. This is an example of why the type MyFace needs to know the type MyVertex

note: Although we left the STL type of container of vertices and faces as a template parameter, at the current state many kernel algorithms of VCG Lib assumes they are STL vector, so if you pass a std::list or a map your use of the library will be quite limited.

How to create a mesh

Once you declared your mesh type, you may want to instance an object and to fill it with vertexes and triangles. It may cross your mind that you could just make some push_back on the vertexes and faces container (data member vert and face of class vcg::tri::Trimesh). In fact this is the wrong way since there can be side effects by adding element to a container. We describe this issue and the correct way of adding mesh element in the Allocation page.

The flags of the mesh elements

Usually to each element of the mesh we associate a small bit vector containing useful single-bit information about vertices and faces. For example the deletion of vertex simply mark a the Deletion bit in thsi vector (more details on the various deletion/allocation issues in the Allocation page. More details on the various kind of flags that can be associated are in the Flags page.

How to process a mesh

The algorithms that *do something* on a mesh are generally written as static member functions of a class templated on the mesh type. For example the code snipped below is part of the class UpdateNormals, which contains the several algorithms to compute the value of the normal

/** vcg/complex/trimesh/update/normal.h */ ... template <class ComputeMeshType> class UpdateNormals{ ... /// Calculates the vertex normal (if stored in the current face type) static void PerFace(ComputeMeshType &m) /// Calculates the vertex normal. Without exploiting or touching face normals /// The normal of a vertex v is the weigthed average of the normals of the faces incident on v. static void PerVertex(ComputeMeshType &m) /// Calculates both vertex and face normals. /// The normal of a vertex v is the weigthed average of the normals of the faces incident on v. static void PerVertexPerFace(ComputeMeshType &m) ... };

This class is part of a kernel of classes with name Update*Value* that compute the value of the vertex or face attributes and that can be found altogether in the folder vcg/complex/trimesh/update. For example, the following example show how to compute the value of the normal and the mean and gaussian curvature per vertex:

/* apps/sample/trimesh_base/trimesh_definition.h */ #include <vector> #include <vcg/simplex/vertex/base.h> #include <vcg/simplex/vertex/component.h> #include <vcg/simplex/face/base.h> #include <vcg/simplex/face/component.h> #include <vcg/complex/trimesh/base.h> **#include <vcg/complex/trimesh/update/normals.h>** //class UpdateNormals **#include <vcg/complex/trimesh/update/curvature.h>** //class UpdateCurvature class MyEdge; class MyFace; class MyVertex: public vcg::VertexSimp2<MyVertex,MyEdge,MyFace, vcg::vertex::Coord3d, **vcg::vertex::Normal3f**,**vcg::vertex::Curvaturef**>{}; class MyFace: public

vcg::FaceSimp2<MyVertex,MyEdge,MyFace, vcg::face::VertexRef>{}; class MyMesh: public vcg::tri::TriMesh< std::vector<MyVertex>, std::vector<MyFace> > {}; int main() { MyMesh m; // fill the mesh ... // compute the normal per-vertex -> update the value of v.N() forall v (vcg::vertex::Normal3f) vcg::tri::UpdateNormals<MyMesh>::**PerVertexPerFace**(m); // compute the curvature per-vertex -> update the value of v.H() and v.K() (vcg::vertex::Curvaturef)
vcg::tri::UpdateCurvature<MyMesh>::**MeanAndGaussian**(m); return 0; }
Other than algorithms that update values of the mesh attributes, VCG Lib provides algorithms to create a mesh from another source, for example from point sets (by means of the Ball Pivoting apprach) or as isosurfaces from a volumetric dataset (by means of Marching Cubes algorithm). Those algorithm can be found in vcg/complex/trimesh/create/.

Finally, you can find algorithms for refinement (midpoint, Loop, Butterfly...), for smoothing, for closing holes and other that are not currently classified under any specific heading and that you can find under /vcg/complex/trimesh.

Optional Component
There are many cases where some vertex or face attributes are not necessary all the time. VCG Lib gives you a way to specify *optional components*, i.e. attributes that are not statically stored within the simplex but can be dynamically allocated when you need them. We use the term *component* to indicate that they are part of the simplex type and that there is a member function to access their value.
To 'define' optional component you need to do two things:

- to use a special type of container (derived from std::vector)
- to specify the right type of component in the template parameters

An optional component can be accessed when the memory for it has been allocated, in which case we say that the component is 'enabled'. An optional component can be enable by calling the function Enable.
VCG Lib handles optional components with two alternative mechanisms: the first is called 'Ocf' (for Optional Component Fast) which uses one pointer for each simplex but it makes accessing optional attribute almost as fast as non-optional ones; the second is called 'Occ' (for Optional Component Compact) which only requires a little extra space for each mesh (unrelated to its size in terms of number of simplicies) but which can result in a slower access. In the following, only their use is discussed. The implementation detail can be found here.

Optional Component Fast
The following definition of MyMesh specifies that the 'normal' component of the type MyVertex is optional. The differences from the previous example are shown in bold.

- include the header for the optional_component_ocf.h which contains the definitions for the optional attributes;
- include the special attribute <vcg::vertex::Info> as first attribute of the type vcg::vertex::VertexSimp2;
- use the type vcg::vector_ocf as the container of your vertices;
- include the type **vcg::vertex::Normal3fOcf** among the template parameters

/* check apps/sample/trimesh_optional/trimesh_optional.cpp */ #include <vector> #include <vcg/simplex/vertex/base.h> **#include <vcg/simplex/vertex/component_ocf.h>** #include <vcg/simplex/face/base.h> #include <vcg/simplex/face/component.h> #include <vcg/complex/trimesh/base.h> class MyEdge; class MyFace; class MyVertex: public vcg::VertexSimp2<MyVertex,MyEdge,MyFace,**vcg::vertex::InfoOcf**,vcg::vertex::Coord3d, **vcg::vertex::Normal3fOcf**>{}; class MyFace: public vcg::FaceSimp2<MyVertex,MyEdge,MyFace,vcg::face::VertexRef>{}; class MyMesh: public vcg::tri::TriMesh<**vcg::vertex::vector_ocf<MyVertex>**, std::vector<MyFace> >{}; int main() { MyMesh m; // ...fill the mesh... MyMesh::VertexIterator vi = m.vert.begin(); (*vi).N() = vcg::Point3f(1.0,1.0,1.0); // ERROR **m.vert.EnableNormal()**; // this allocate the memory to store the normal (*vi).N() = vcg::Point3f(1.0,1.0,1.0); // OK **m.vert.DisableNormal()**; // this deallocate the memory to store the normal (*vi).N() = vcg::Point3f(1.0,1.0,1.0); // ERROR (again)! return 0; }

Before accessing the data contained in the *normal* component, you must enable the attribute by calling EnableNormal(), afterwards the space for the component will be allocated and accessible until you call DisableNormal(). Trying to access the value of the normal before enabling or after disabling the corresponding component will throw an assertion.

Optional Component Compact

The following definition of MyMesh specifies that the 'normal' attribute of the type MyVertex is optional. To use a Occ component you must:

- include #include <vcg/simplex/vertex/component_occ.h;
- use the type vcg::vector_occ as the container of your vertices.
- specify the attribute normal as optional by passing **vcg::vertex::Normal3fOcc** among template parameters of VertexSimp2.

/* check apps/sample/trimesh_optional/trimesh_optional.cpp */ #include <vector> #include <vcg/simplex/vertex/base.h> **#include <vcg/simplex/vertex/component_occ.h>** #include <vcg/simplex/face/base.h> #include <vcg/simplex/face/component.h> #include <vcg/complex/trimesh/base.h> class MyEdge; class MyFace; class MyVertex: public vcg::VertexSimp2<MyVertex,MyEdge,MyFace, vcg::vertex::Coord3d, **vcg::vertex::Normal3fOcc**>{}; class MyFace: public vcg::FaceSimp2<MyVertex,MyEdge,MyFace,vcg::face::VertexRef>{}; class MyMesh: public vcg::tri::TriMesh< **vcg::vertex::vector_occ<MyVertex>**, std::vector<MyFace> >{}; int main() { MyMesh m; // ...fill the mesh... MyMesh::VertexIterator vi = m.vert.begin(); (*vi).N() = vcg::Point3f(1.0,1.0,1.0); // ERROR **m.vert.Enable<vcg::vertex::NormalType>()**; // this allocate the memory to store the normal (*vi).N() = vcg::Point3f(1.0,1.0,1.0); // OK **m.vert.Disable<vcg::vertex::NormalType>()**; // this deallocate the memory to store the normal (*vi).N() = vcg::Point3f(1.0,1.0,1.0); // ERROR (again)! return 0; }

Before accessing the data contained in the normal, you must enable the attribute by calling EnableAttribute<vcg::vertex::Normal3fOcc::NormalType>(),

afterwards the attribute will be allocated and accessible until you call DisableAttribute<vcg::vertex::Normal3fOcc::NormalType>(). Trying to access the value of the normal before enabling or after disabling the corresponding attribute will throw an assertion.
Two important things about optional optional components:

- The access function to the value of a component is the same both if the component is optional and if it's not. This means that you do not have to make this distinction when coding your algorithm
- If you make a copy of the vertex(face) and then try to access a component it won't work. The mechanisms for optional attributes works using the position of the vertex(face) in memory. In other words, the optional data is associated with a position of the container (vector_ocf or vector_occ) and not bound to the vertex itself.

User-defined attributes
VCG Lib also provides a simple mechanism to associate user-defined 'attributes' to the simplicies and to the mesh. Note that both 'attributes' and 'components' are basically accessory data that are bound to a simplex. Conceptually the difference is that with the term component VCGLib indicates those values that are considered to 'define' the simplex (its position, its normal its connectivity information), while the user defined attribute is an accessory data which make sense to some specific algorithm, like "the number of time a vertex has been moved" or "a pointer to a string containing a description of the vertex".
Practically the difference is that every optional component has its non optional counterpart and is accessed through a member function of the simplex, so that when you write your algorithm you use v.N() to access the normal both it is has been declared as optional or not, while the attributes are accessed by a handle which is returned at the creation of the attribute.

The following code snippet shows an example:

```
/* apps/sample/trimesh_attribute/trimesh_attribute.cpp */
#include<vcg/simplex/vertex/base.h>
#include<vcg/simplex/vertex/component.h>
#include<vcg/simplex/face/base.h> #include<vcg/simplex/face/component.h>
#include<vcg/complex/trimesh/base.h>
#include<vcg/complex/trimesh/allocate.h> class MyFace; class MyVertex; class MyEdge; // dummy prototype never used class MyVertex : public vcg::VertexSimp2< MyVertex, MyEdge, MyFace, vcg::vertex::Coord3f,vcg::vertex::Normal3f>{}; class MyFace : public vcg::FaceSimp2< MyVertex, MyEdge, MyFace, vcg::face::VertexRef, vcg::face::Normal3f> {}; class MyMesh : public vcg::tri::TriMesh< std::vector<MyVertex>, std::vector<MyFace> > {}; float Irradiance(MyMesh::VertexType v){ // ..... return 1.0; } int main() { MyMesh m; //...here m is filled // add a per-vertex attribute with type float named "Irradiance" MyMesh::PerVertexAttributeHandle<float> ih = vcg::tri::Allocator<MyMesh>::AddPerVertexAttribute<float>
(m,std::string("Irradiance")); // add a per-vertex attribute with type float named "Radiosity"
vcg::tri::Allocator<MyMesh>::AddPerVertexAttribute<float>
```

(m,std::string("Radiosity")); // add a per-vertex attribute with type bool and no name specified MyMesh::PerVertexAttributeHandle<bool> blocked_h = vcg::tri::Allocator<MyMesh>::AddPerVertexAttribute<bool> (m); MyMesh::VertexIterator vi; int i = 0; for(vi = m.vert.begin(); vi != m.vert.end(); ++vi,++i){ ih[vi] = Irradiance(*vi); // [] operator takes a iterator ih[*vi] = Irradiance(*vi); // or a MyMesh::VertexType object ih[&*vi]= Irradiance(*vi); // or a pointer to it ih[i] = Irradiance(*vi); // or an integer index } // Once created with AddPerVertexAttribute, an handle to the attribute can be obtained as follows MyMesh::PerVertexAttributeHandle<float> rh = vcg::tri::Allocator<MyMesh>::GetPerVertexAttribute<float>(m,"Radiosity"); // you can query if an attribute is present or not bool hasRadiosity = vcg::tri::HasPerVertexAttribute(m,"Radiosity"); // you can delete an attibute by name vcg::tri::Allocator<MyMesh>::DeletePerVertexAttribute<float>(m,"Radiosity"); // you can delete an attibute by handle vcg::tri::Allocator<MyMesh>::DeletePerVertexAttribute<bool>(m,blocked_h); }

The same can be done for the faces, just replace the occurrences of PerVertex with PerFace. Note that if you call add an attribute *without* specifying a name and you lose the handle, you will not be able to get your handle back.

Note:

- Do not get mix up the scope of the handle with the memory allocation of the attribute. If you do not delete an attribute explicitly, it will be allocated until the mesh itself is destroyed, even if you do not have handles to it.

C++ type of a mesh and reflection

VCG Lib provides a set of functions to implement reflection, i.e. to investigate the type of a mesh at runtime. These functions follow the format Has[attribute](mesh) and return a boolean stating if that particular attribute is present or not.

template<class ComputeMeshType> static void UpdateNormals<ComputeMeshType>::PerVertex(ComputeMeshType &m) { if(!HasPerVertexNormal(m)) return; ... }

You may wonder why those functions are not statically typed and why they needs the mesh object, i.e. why can't you just write ComputeMeshType::HasPerVertexNormal()? The reason is that VCG Lib reflection takes into account optional components, therefore HasPerVertexNormal(m) will return true if the type of the vertex contains the attribute as permanent (e.g. vcg::vertex::Normal3f) OR if it contains the attribute as optional (e.g. vcg::vertex::Normal3fOcf) AND it is enabled, i.e. the relative Enable function has been called.

Adjacency

VCG Lib does not have a hard-coded way to encode the adjacencies among simplices. It all depends on which attributes are stored with the simplices and how they are used. I the previous examples the definition of face has always included the attribute vcg::face::VertexRef , which stores 3 pointers to MyVertex

accessible with the member function V() (the well known Indexed Data Structure). The reason is that almost all of the algorithms currently implemented in VCG Lib assume its presence. So, if your type MyFace does not include the attribute vcg::face::VertexRef, the definition will be correct but almost no algorithm will work. There are other adjacency relations that can be useful to *navigate* a mesh, for example to collect the one-ring neighborhood of a vertex. VCG Lib uses two mechanisms which are explained in the following, along with the attributes they use.

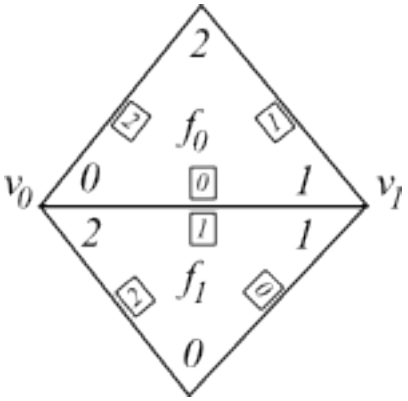
FF Adjacency

The face-to-face adjacency, stored in the attribute for the faces vcg::face::FFAdj (vcg::face::TTAdj for tetrahedra), encodes the adjacency of faces (tetrahedra) through edges (faces). The image below shows two triangle faces with the convention adopted for indexing vertexes and edges.

The vertexes are numbered from *0* to *2* in CCW sense and the edge *i = 0..2* is the edge whose extremes are *i* and *(i+1) mod 3* . Therefore the common edge between faces f_1 and f_2 is the edge *1* of the face f_1 and the edge *0* of the face f_0.

vcg::face::FFAdj stores, for each edge *e* of the face *f*:

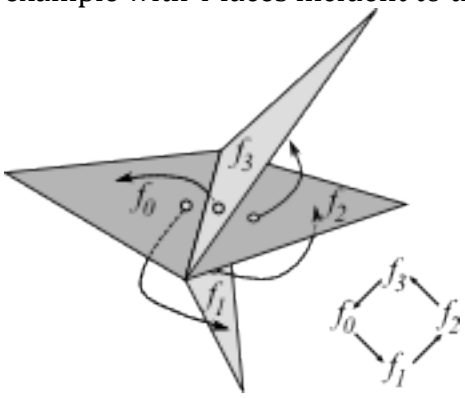
- FFp(e): a pointer to **a** face *f'* sharing *e*. If *e* is a border than points to the face *f* itself
- FFi(e):the index of *e* in the pointed face
-



f1->FFp(1) == f0
f1->FFi(1) == 0
f0->FFp(0) == f1
f0->FFi(1) == 1

Indexing of vertexes and edges

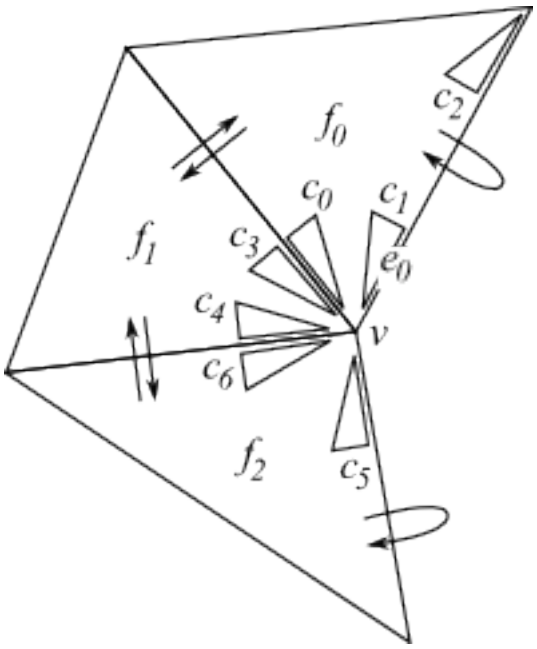
Note that we specified that FFp(e) point to **a** adjacent face, not to **the** adjacent face. The reason is that there could be more than two faces on the same edge, and this is seamlessly supported by VCG Lib. In the picture below is shown an example with *4* faces incident to the same edge



Non manifold edge

In this case the adjacencies are set to form a circular list (not necessarily sorted with the angle around the edge). This is done by the VCG Lib function that update these values (UpdateTopology<MeshType>::FFTopology(MeshType & m)).
In this way VCG Lib provides a way to check if a mesh is manifold on a specific edge because the FF adjacency relation is **mutual**, i.e. the face f_0 points to the face f_1 which points to the face f_0, if and only if the mesh is manifold over the corresponding edge.
bool IsManifold(MyFace *f,int e) { return (f == f->FFp(e)->FFp(f->FFi(e)))}
Referring to the picture:
(f0 == f0->FFp(0)->FFp(f0->FFi(0))} equals
(f0 == f1->FFp(0)} equals
(f0 == f0} Ok! It is manifold

Pos
The Pos is the VCG Lib implementation of the Cell-Tuple{ref}. Here we give a as short as possible definition, trading formalisms for space. A Pos in a triangle mesh is a triple made of a vertex: *pos = (v,e,f)* , such that *v* is an extreme of *e* and *e* belong to the face *f* . The figure shows few pos in a triangle mesh as small triangles "pointing" to a vertex, "leaning" against an edge and inside a face. For example *c0=(v,e0,f)*.
A very nice property is that given a pos *c*, there is only another *neighbor* pos *c'* that can be obtained from *c* changing only one of the elements of the triple.
We call the operation of passing from a pos to one of its neighbors *Flip* and write **FlipV**, **FlipE** and **FlipF** to indicate that the *flipped* element is the vertex, the edge or the face respectively.
For example consider *c1*: there is only another pos which is the same as *c0* except for the vertex component of the triple, and it is *c2*. For brevity, we write *c2 = FlipV(c1)*.
In the left of the table some other examples are shown just to make this point clear.



c2 = Flip**V**(c1)
c0 = Flip**E**(c1)
c3 = Flip**F**(c0)

CCW around v
c4 = Flip**E**(Flip**F**(c0))
c5 = Flip**E**(Flip**F**(c4))
Bounce
c6 = Flip**E**(Flip**F**(c5))
CW around v
c3 = Flip**E**(Flip**F**(c6))
c1 = Flip**E**(Flip**F**(c3))
Bounce
c0 = Flip**E**(Flip**F**(c1))

Note that concatenating two flips: FlipF and FlipE we obtain a transition from a pos to the next in counterclockwise or in clockwise sense, depending if the starting pos is on the CCW edge of the face with respect to the vertex or not. Also note that, thanks to how FF adjacency is defined, when a pos is on the border, it *bounces* back. This pair of flips are widely used in the VCG Lib to run over the one ring neighborhood of manifold vertices.

The following code snippet shows how to use the pos to iterate around a vertex:
/* vcglib/apps/sample/trimesh_pos_demo/trimesh_pos_demo.cpp */ #include <vcg/simplex/face/pos.h> // include the definition of pos //...includes to define your mesh type //class MyVertex: ... class MyFace: public vcg::FaceSimp2<MyVertex,MyEdge,MyFace,
vcg::face::VertexRef, **vcg::face::FFAdj**>{}; void OneRingNeighborhood(MyFace * f) { MyVertex * v = f->V(0); MyFace* start = f; vcg::face::Pos<MyFace> p(f,0,v);// constructor that takes face, edge and vertex do { p.FlipF(); p.FlipE(); }while(p.f!=start); }

Two important notes:

- We arbitrarily picked f->V(0) as pivot vertex. In general one may want to start knowing the vertex. This is done by including the attribute vcg::vertex::VFAdj which basically associates to each vertex pointer to one of the faces incident on it. See the VF Adjacency for details.
- This implementation does not work if the vertex is on the border. Just try with the example: from the pos *c4* it would find *c5*,*c6*,*c3* which is in the same face as *c4*. Of course this does not happen if you use the pos itself as a guard and not just the face. However, even in this case, you would obtain the sequence of pos: *c5*,*c6*,*c3*,*c1*,*c0*,*c4* corresponding to the faces *f2*,*f2*,*f1*,*f0*,*f0*,*f1* which probably is not what you want. VCG Lib provides a variation of pos that solves this problem

Jumping Pos

The Jumping Pos works exactly like the Pos, only it does not bounce when it encounters the border. Instead, it jumps *around* the shared vertex of the border-faces as if they were adjacent (faces *f0* and *f2* in the image).
/* vcglib/apps/sample/trimesh_pos_demo/trimesh_pos_demo.cpp */ #include <vcg/simplex/face/jumping_pos.h> // include the definition of jumping pos //...includes to define your mesh type //class MyVertex: ... class MyFace: public vcg::FaceSimp2<MyVertex,MyEdge,MyFace,
vcg::face::VertexRef,vcg::face::FFAdj>{}; void OneRingNeighborhoodJP(MyFace * f) { MyVertex * v = f->V(0); MyFace* start = f; vcg::face::JumpingPos<MyFace> p(f,0,v);// constructor that takes face, edge and vertex do { p.NextFE(); }while(p.f!=start); }

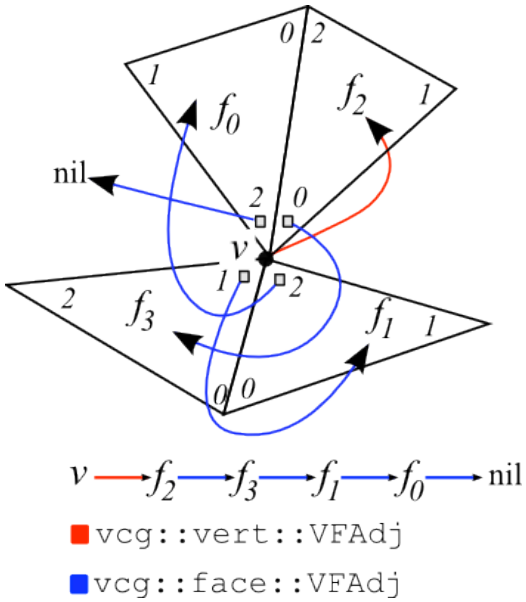
VF Adjacency

VCG Lib implements vertex-to-face adjacency, i.e. given a vertex *v* we can retrieve all the faces incident to *v*. Let *v_star =(f0,f1,f2,...,fk)* be the set faces incident to *v* arranged in a sequence (with no preferred criteria). VCG Lib allows to retrieve *v_star* in optimal time (***O****(#star_v)* by using the following attributes:

- vcg::vertex::VFAdj which is a vertex attribute containing a pointer to *f0*

- vcg::face::VFAdj which is a face attribute containing a pointer to the next face in the list *v_star* for each of its 3 vertices (4 in the case of tetrahedra)

These two attributes are not only pointers, they also contain an index referring the index of the vertex in the pointed face in the same style as the vcg::face::FFAdj does. The picture below shows a complete example:



v.VFp()==f2
v.VFi() == 0
f2->VFp(0)==f3
f2->VFi(0) == 1
f3->VFp(1)==f1
f3->VFi(1) == 2
f1->VFp(2)==f0
f1->VFi(2) == 2
f0->VFp(2)==NULL
f0->VFi(2) == -1

example of vertex-face adjacency

VFIterator

VFIterator is a simple iterator to run over the faces in the one-ring neighborhood of a vertex using the VF Adjacency (it is just like Pos for the FF Adjacency) The following code snippet shows how to use the VFIterator:

```
/* vcglib/apps/sample/trimesh_pos_demo/trimesh_vfiter_demo.cpp */
#include <vcg/simplex/face/pos.h> // include the definition of VFIterator
//...includes to define your mesh type class MyVertex: public vcg::VertexSimp2<MyVertex,MyEdge,MyFace, vcg::vertex::VFAdj /*,... other attributes*/ >{}; class MyFace: public vcg::FaceSimp2<MyVertex,MyEdge,MyFace, vcg::face::VertexRef,vcg::face::VFAd>{}; void OneRingNeighborhoodVF( MyVertex * v) { vcg::face::VFIterator<MyFace> vfi(v); //initialize the iterator tohe first face for(;!vfi.End();++vfi) { MyFace* f = vfi.F(); // ...do something with face f } }
```

Few facts on FF adjacency and VF adjacency

Here we make a series of simple statements just to avoid confusion and try to help up choosing the adjacencies the best fit your needs.

- If the mesh is manifold, the one-ring neighborhood of a vertex computed by using Pos (needs FF adjacency) is the same as the one computed by using VFIterator (needs VF adjacency). The order in which the faces are visited can be CW or CCW if using Pos, unspecified by using VFIterator

- If the mesh is non-manifold, Pos may not find all the faces of the one-ring neighborhood of the vertex, VFIterator always does

Boundary relations and adjacency
In many algorithms you need to simply the boundary/border condition of a face, e.g. to know if a given face f has one or more adjacent faces on a specified edge e. Using FF adjacency this can be done simply by using the face::IsBorder(f,e) static function that simply checks if the pointer stored in face f on the edge e points to f itself. If you are navigating the mesh using a Pos, you have a Pos member function IsBorder() that reports the boundary condition of the current pos. Similarly, for testing manifoldness of specific places over a mesh, there is a face::IsManifold(f,e) static function and a IsManifold(e) function member of the pos class.

If you are not using FF adjacency evaluating the boundary conditions could be not very efficient, so vcg library provides a technique to *cook* the current boundary conditions of the mesh into vertex and face flags. Use the members of the UpdateFlags static class to compute flags that reflects the current mesh status and the access these flags using the IsB(e) member function of the face class. Remember that flags based boundary information can become invalid if you change the mesh topology. On the other hand consider that many non-mesh-modifying algorithms do not require explicit FF adjacency but just boundary information (typical examples: most mesh smoothing and curvature computation algorithms).

Please note that the boundary flags are set true also for non manifold conditions.

Space concepts
VCG Lib implements the basic functionalities to handle geometric entities such as the point, the segment, the triangle and so on. It is very important not to confuse those geometric entities and the topological entities they can be a component of. In other words, the type vcg::Point3f (vcg/space/point3.h) is a point in 3 dimensional space and is different from a vertex (vcg::VertexSimp2<...>) which is a topological entity. The same goes for the vcg::Segment3 and the vcg::EdgeSimp2<..>, the vcg::Triangle3 and the vcg::FaceSimp2.

Viewing and manipulation
In this section, all that concerns the definition of a view and the manipulation of a mesh will be explained.

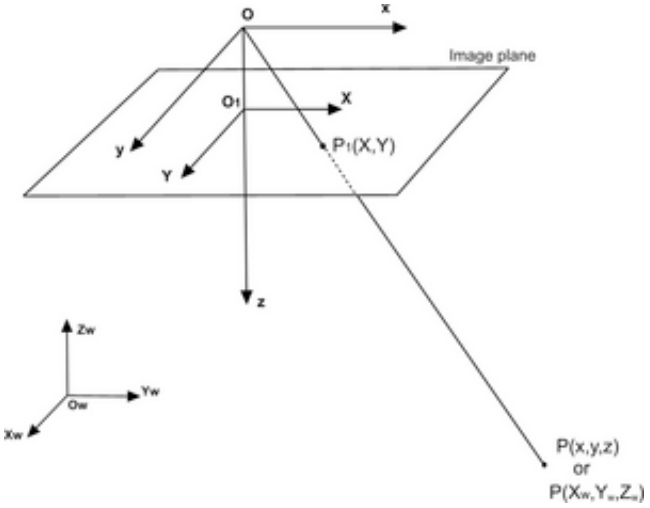
Shot and camera
This section presents the structure of the shot and camera of the library. After an overview of the camera model used, all the components of the shot class are listed and described. Then, a set of examples of the most important operations (projection and un-projection) and of the interactions between shots and between a shot and the trackball are presented. Finally, simple examples of a shot are visually shown, in order to help with the implementations of eventual wrapped to and from other shot-camera formats.

The camera model

In general, the camera parameters can be divided in two groups:

- **Extrinsic (or external) parameters**: these are the parameters associated to the position in the space of the camera.
- **Intrinsic (or internal) parameters**: these values are related to the peculiar characteristics of the camera, like the focal length (the *zoom*) or the distortion introduced by the lenses.

If these groups of values are put in a proper camera model, it is possible to transform any point in the space in the corresponding point on the image plane of the camera (and *vice versa*).



An example scheme of a perspective camera model

In fact, given a simple perspective camera model like the one shown in figure, extrinsic parameters can be used to transform a point from its world coordinates $\{x_w,y_w,z_w\}$ to the camera 3D coordinate system {x,y,z}):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} + T$$

In this case the extrinsic parameters are a 3 X 3 rotation matrix R and a translation vector T, which define the *orientation* and *position* of the camera. In order to transform the 3D camera coordinate in 2D image plane coordinates (X_u,Y_u) it's necessary to know the measure of the distance between the point of view and the image plane (OO_1 in figure): this value, indicated with *f*, is usually known as the *focal length*. The relation between the camera and image coordinates of the point can be expressed as follows:

$$X_u = f\frac{x}{z} \quad , \quad Y_u = f\frac{y}{z}$$

Another aspect of the structure of a camera that can be characterized is the distortion introduced by the lenses: if we suppose that the distortion is radial (performed along the radial direction respect to the center of distortion) we can calculate the undistorted image coordinates

$$X_d + D_x = X_u \ , \ Y_d + D_y = Y_u$$

where

$$D_x = X_d(k_1 r^2 + k_2 r^4 + ...) \ , \ D_y = Y_d(k_1 r^2 + k_2 r^4 + ...)$$

and

$$r = \sqrt{X_d^2 + Y_d^2}$$

In conclusion, a quite accurate model of a camera can be described by:

- A 3x3 rotation matrix and a translation vector for extrinsic parameters
- The values of focal, center of distortion and one or more distortion coefficients for intrinsic parameters

While this set of parameters provides everything to transform any 3D point in its corresponding point in the image plane, this could be not enough in peculiar applications. If an accurate estimation of the real position of the camera respect to the object is needed, some more data about the camera model are needed: in particular, the sensor physical size together with the resolution in pixel of the acquired image. If these a-priori data are known, a unique set of camera parameters is associated to any shot.

The VCG Shot

The implementation of a Shot in the VCG library can be found in vcg\math\shot.h
The shot is composed by two elements:

- the Extrinsics parameters, which are stored in the class Shot (in the type ReferenceFrame) that contains viewpoint and view direction.

The Extrinsics parameters are kept as a rotation matrix "rot" and a translation vector "tra" NOTE: the translation matrix "tra" corresponds to -viewpoint while the rotation matrix "rot" corresponds to the axis of the reference frame by row, i.e.
rot[0][0 1 2] == X axis
rot[1][0 1 2] == Y axis
rot[2][0 1 2] == Z axis

It follows that the matrix made with the upper left 3x3 equal to rot and the 4th column equal to tra and (0,0,0,1) in the bottom row transform a point from world coordinates to the reference frame of the shot.

- the Instrinsics parameters, which are stored as a Camera type (check vcg/math/camera) and that determines how a point in the frame of the camera is projected in the 2D projection plane. This information was kept indendent of the extrinsic parameters because more than one shot can share the same intrinsic parameters set.

The attributes of a Camera, which is define in vcg\math\shot.h, are:

//------ camera intrinsics ScalarType FocalMm; /// Focal Distance: the distance between focal center and image plane. Expressed in mm Point2<int> ViewportPx; /// Dimension of the Image Plane (in pixels) Point2< S> PixelSizeMm; /// Dimension in mm of a single pixel Point2< S> CenterPx; /// Position of the projection of the focal center on the image plane. Expressed in pixels Point2< S> DistorCenterPx; /// Position of the radial distortion center on the image plane in pixels S k[4]; /// 1st & 2nd order radial lens distortion coefficient (only the first 2 terms are used) //----------

While the extrinsic parameters usually change between the shots, some (sometimes all) of the intrinsic are strongly related to the camera model used. In particular, some values are usually known before camera calibration: viewportPx and CenterPx. Moreover, if an accurate calibration is needed, it is necessary to fill the PixelSizeMm value. This can be inferred from the camera datasheet (it can be calculated by dividing the sensor width and height, in mm, by the resolution of the image) or, in some cases, from the EXIF of the image (in the CANON models, it is the inverse of FocalPlane X-resolution and FocalPlane Y-resolution, scaled from inches to mm if necessary). If a correct PixelSizeMm is not set, the values of the camera parameters can be different from the real ones, even though the image is perfectly aligned to a 3D model. Also the focal distance can be inferred from EXIF, but its value is indicative. If a calibrated camera is used, then all the intrinsic parameters should be known in advance.

File Formats

VCGLib provides importer and exporter for several file formats

- import: PLY, STL, OFF, OBJ, 3DS, COLLADA, PTX, V3D, PTS, APTS, XYZ, GTS, TRI, ASC, X3D, X3DV, VRML, ALN
- export: PLY, STL, OFF, OBJ, 3DS, COLLADA, VRML, DXF, GTS, U3D, IDTF, X3D

The following code snippet show how to use the PLY importer and exporter:

#include <wrap/io_trimesh/import.h> // definition of type MyMesh MyMesh m; vcg::tri::io::ImporterPLY<MyMesh>::Open(m,"namefile_to_open.ply"); // .... vcg::tri::io::ExporterPLY<MyMesh>::Save(m,"namefile_to_save.ply");

The previous one is the minimal required interface for all the importer and exporters. Additionally two other parameters can be specified: mask and callback . The callback is used for providing a feedback during the usually lenght importing/exporting process. The mask is used to better specify/understand what is loaded/saved.

Saving Mask and Reading Mask

Generally all file formats save at least vertex positions and basic connectivty information, but beyond that you may want to choose which other data are stored with the file. To this aim, VCGLib provides a class vcg::tri::io::Mask which is essentially a collection of bit names that are used to specify which components you want to write to the file (e.g. stuff like vcg::tri::io::Mask::IOM_VERTCOLOR).

When **saving** this mask is used in a *READ ONLY* mode (they are just constants) to specify what component you want to save. For example, if you have stored in your mesh normals per vertex (for rendering purposes) but you consider saving them in ply format just a waste of space, you should specify an appropriate mask with the Mask::IOM_VERTNORMAL bit cleared. Obviously not all the formats are able to save all the possible data. For example STL format cannot save texture coords or per vertex color. So to know what you are able to save in a given XXX format there is the function ExporterXXX::GetExportMaskCapability() that gives you a bitmask with all the capability of that file format.

When **loading** this mask is used in a *WRITE ONLY* mode to report to the user what fields have been loaded from the file into the mesh (e.g. the initial value of the mask is ignored). In many cases it could be useful to know what data is present into a file to *prepare* the mesh (e.g. to enable optional components), for this purpose all the importer have also a ImporterXXX::LoadMask(filename,mask) that fill the mask only without effectively loading a mesh. Note that for some mesh formats understanding what is contained into a file means parsing the whole file.

Error Reporting

The mesh I/O functions returns ZERO on success and an error code different from zero in case of failure or critical conditions. Use the static const char *ErrorMsg(int error) function to get a human understandable description of the error code and static bool ErrorCritical(int err) to know if a given error is a critical one or just a warning. For example in the OBJ format the absence of the accompanying material description file, is considered non critical (you are able to get the correct geometry even if you miss material info).

VMI dump file

VMI is an acronym for **V**cglib **M**esh **I**mage and it is **not** a proper file format but a simple dump of the vcg: TriMesh on a file. Not being a file format means that:

- there is no specification
- it is bound to the current version of the VCGLib with no backward compatibility

a VMI can be very useful for debug purpose, because it saves:

- all the components
- all the temporary components (currently only of Optional Component Fast type)
- all the vertex, face or mesh attributes you have in your mesh file

So if, for example, your algorithm crashes at the n-th step, you can save intermediate results of your computation with VMI and reload it.

Note that in order to load a VMI the mesh passed as template to vcg::tri::ImporterVMI<MeshType>(..) must be of the same type as the mesh passed to vcg::tri::ExporterVMI<MeshType>(..) otherwise the loader returns FALSE.

Access to mesh

Assumption: the mesh is stored in a object m of type MyMesh

Accessing the coords of all the vertexes

```
MyMesh::VertexIterator vi; for(vi = m.vert.begin(); vi!=m.vert.end(); ++vi )  {
DoSomething((*vi).P());  }
```

Accessing all the faces and computing their barycenter

```
MyMesh::FaceIterator fi;  for(fi = m.face.begin(); fi!=m.face.end(); ++fi )  {
MyMesh::CoordType b = ((*fi).V(0)->P() + (*fi).V(1)->P() + (*fi).V(2)->P() ) / 3.0;
}
```

or using a common shorthand for accessing to the coord of a face vertex:

```
MyMesh::FaceIterator fi; for(fi = m.face.begin(); fi!=m.face.end(); ++fi )  {
MyMesh::CoordType b = ((*fi).P(0) + (*fi).P(1) + (*fi).P(2) ) / 3.0;  }
```

Or even in a shorter way:

```
MyMesh::FaceIterator fi; for(fi = m.face.begin(); fi!=m.face.end(); ++fi )  {
MyMesh::CoordType b = vcg::Barycenter(*fi);  }
```

Creating elements

Creating the simplest single triangle mesh.

```
m.Clear(); Allocator<MyMesh>::AddVertices(m,3); Allocator<MyMesh>::AddFaces(m,1); MyMesh::VertexPointer ivp[3]; VertexIterator vi=in.vert.begin(); ivp[0]=&*vi;(*vi).P()=CoordType ( 1.0, 1.0, 1.0); ++vi; ivp[1]=&*vi;(*vi).P()=CoordType (-1.0, 1.0,-1.0); ++vi; ivp[2]=&*vi;(*vi).P()=CoordType (-1.0,-1.0, 1.0); ++vi; FaceIterator fi=in.face.begin(); (*fi).V(0)=ivp[0]; (*fi).V(1)=ivp[1]; (*fi).V(2)=ivp[2];
```

look to complex/trimesh/create/platonic.hplatonic.h for more examples.

Destroying Elements

Lazy deletion strategy.

Note that the two basic deletion strategies are very low level functions. They simply mark as deleted the corresponding entries without affecting the rest of the structures. So for example if you delete a vertex with these structures without checking that all the faces incident on it have been removed you create a non consistent situation. Similarly, but less dangerous, when you delete a face its vertices are left around so at the end you can have unreferenced floating vertices.

Allocator<MyMesh>::DeleteVertex(m,v); Allocator<MyMesh>::DeleteFace(m,v);

If your algorithm performs deletion the size of a container could be different from the number of valid element of your meshes (e.g.:

m.vert.size() != m.vn m.face.size() != m.fn

Therefore when you scan the containers of vertices and faces you could encounter deleted elements so you should take care with a simple !IsD() check:
MyMesh::FaceIterator vi; for(fi = m.face.begin(); vi!=m.face.end(); ++fi) if(!(*fi).IsD()) // <---- Check added { MyMesh::CoordType b = vcg::Barycenter(*fi); }

In some situations, particularly when you have to loop many many times over the element of the mesh without deleting/creating anything, it can be practical and convenient to get rid of deleted elements by explicitly calling the two garbage collecting functions:
Allocator<MyMesh>::CompactVertexVector(m);
Allocator<MyMesh>::CompactFaceVector(m);

After calling these function it is safe to not check the IsD() state of every element and always holds that:
m.vert.size() == m.vn m.face.size() == m.fn

Note that if there are no deleted elements in your mesh, the compactor functions returns immediately.

Adjacency relations
VCG meshes DO NOT store edges, only vertex and triangles. Even the basic adjacency relations have to be explicitly computed. We always try to do not store per-edge information and we try to keep such information only more or less implicitly. This is simply because there are many algorithms that do not explicitly require this information and the cost of updating/storing should be avoided when possible. In many cases algorithms that seems based on some kind per-edge iteration or information can be re-written in a more efficient way by moving the loop to other mesh elements.

Counting border edges (without topology)
Typical example are border flags. Each face keep a bit for each of its sides saying if that face has a boundary on that side or not. This information can be accessed by mean of the IsB(i) function. To compute the border flags there are various algorithm according to the available topological informations.
Allocator<MyMesh>::CompactFaceVector(m);
UpdateFlags<MyMesh>::FaceBorderFromNone(m); int BorderEdgeCounter=0; MyMesh::FaceIterator fi; for(fi = m.face.begin(); fi!=m.face.end(); ++fi) { for(int i=0;i<3;++i) if((*fi).IsB(i)) BorderEdgeCounter++; }

Counting border edge (using FF adjacency)
When your mesh as Face-Face adjacency the FFp(i) member store the pointer of the face that is adjacent to the current face along the i-th edg
int BorderEdgeCounter=0; MyMesh::FaceIterator fi; for(fi = m.face.begin(); fi!=m.face.end(); ++fi) { for(int i=0;i<3;++i) if((*fi).FFp(i) == &*fi) BorderEdgeCounter++; }

or alternatively using the same flag based approach as above

```
UpdateFlags<MyMesh>::FaceBorderFromFFAdjacency(m); int BorderEdgeCounter=0; MyMesh::FaceIterator fi; for(fi = m.face.begin(); fi!=m.face.end(); ++fi ) { for(int i=0;i<3;++i) if((*fi).IsB(i)) BorderEdgeCounter++; }
```

Licenses

MeshLab uses some third party open source libraries for its working. Here is the complete lists of the projects that we gratefully thanks for their kind liberality. MeshLab is licensed under the GNU General Public License (GPL).

- VCG Library, developed at the Visual Computing Group - ISTI - CNR. Responsible for all the mesh processing and rendering tasks. (GPL)
- QT, TrollTech the standard framework for high performance, cross-platform application development.
- GLEW: The OpenGL Extension Wrangler Library (BSD)
- lib3ds a software library for managing 3D-Studio Release 3 and 4 ".3DS" files. (LGPL)
- bzip2 a freely available, patent free, high-quality data compressor. (BSD)
- Universal 3D Sample Software Set of libraries to write, read, extend, render and interact with U3D-formatted data, as defined by standard ECMA-363. This library is not directly linked but the U3D plugin invokes the idtfcoverted executable, whose sources are provided with the examples of the library. A binary of the idtfconverter is directly provided with the MeshLab distribution.
- Poisson Surface reconstrution, heavily based on the code kindly provided by Michael Kazhdan and Matthew Bolitho (custom license thanks Misha and Matthew!)
- movie15 Latex package a LaTeX style to embed movies, sounds and 3D objects into PDF documents; the needed glue to generating pdf with embedded U3D objects.
- Some of the editing tools icons came from Tango Icon Library projects.

Privacy Disclaimer

MeshLab will automatically check for the availability of updated versions and will notify the need of upgrading the software to the users. For this reason, from time to time, MeshLab will issue a http network connection. If you prefer that MeshLab does not communicate in any way with its developers, simply use a plain firewall and prevent any MeshLab access to the network. This will not limit in any way the normal behavior of MeshLab (apart from getting notified of new MeshLab releases) Moreover, when using MeshLab, it locally collects some aggregated statistical data about only the specific usage of MeshLab: the overall number and averaged size of the opened/saved meshes; nothing more. Periodically this information is sent back to the developers. This data will be used for statistical analysis and for the assessment of the MeshLab usage. We would like to remark that the kind of information collected by MeshLab is probably by far less sensitive than the information that is silently collected by most web sites when you surf them: IP address, visited pages, frequency of return. We would also like to stress that we really need this information in order to assess how diffuse MeshLab is used and its impact on the 3D community.

Acknowledgments
Some of the MeshLab developers got financial support for their work. For the Italian CNR employees (and for the other involved partners) we acknowledge the financial support of the following projects:

- ***3D-CoForm*** (EU IST): "The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under

grant agreement n° 231809"

- "EPOCH" (EU Network of Excellence, IST-2002-507382)
- MIUR "BLU-ARCHEOSYS", 2006-2009

A fully comprehensive guide on writing an IO Plugin for MeshLab

How to download MeshLab and all the related stuff

Please note that MeshLab uses Qt sdk, so first of all you have to download it from http://www.qtsoftware.com/downloads .

In order to get the last version of MeshLab's code you need (if you haven't already one) a SVN client. We suggest Tortoise SVN. It provides a simple and complete interface to interoperate with a SVN Server.

In the next steps I will assume you have it:

- create every where you want an empty folder and called it meshlab (for instance in *c:\devel\meshlab*)
- right click on the created folder than on the menu clik on *"SVN checkout"* item . On the "url of repository" paste and copy https://meshlab.svn.sourceforge.net/svnroot/meshlab/trunk/meshlab
- at the same level of the previously created meshlab's folder add another folder and called it vcglib (following the assumption of the previous example *c:\devel\vcglib*)
- right click on the created folder than on the menu clik on *"SVN checkout"* item . On the "url of repository" paste and copy https://vcg.svn.sourceforge.net/svnroot/vcg/trunk/vcglib . VCGLib is a geometric and algorithmic library developed in our lab. MeshLab uses it extensively.

How to compile MeshLab and all the related stuff
Now you have succesfully downloaded all you need in order to compile meshlab.
MeshLab uses a few third parts GPL libraries. You will find them in *meshlab/src/external* . In this folder there is also a usefull file called *external.pro* to simply compile all the external dependecies.

If you had correctly installed qt on your pc in the windows start menu you should have a Qt item and in the subitem one labelled *"Qt 4.x command prompt"* . Click on it, a dos shell will appear.

You have to go to folder *meshlab/src/external*. Then write:

qmake -recursive external.pro make

with the last two commands you have compiled the external libraries. Then , in the same dos shell, you have to digit

cd ../ qmake -recursive meshlabv12.pro make

and after waiting a long time (depending on your pc's processor) you have eventually compiled meshlab and all his plugins.

Please note that in the qt sdk package is included a usefull IDE called QtCreator with wich you could avoid to manually digit the previous commands. All you have to do is click on *"file->open"* and import the pro file in it and then click on the *"build"* option.

QtCreator is a good tool but the best way to compile and write code for meshlab maybe is Visual Studio 2008 Express Edition + Qt Visual Studio Integration Tool (you could download it in the "other download" section). In this case you have only to click on *"qt->open solution from .pro file"* to start to work with MeshLab's code.

How to work with MeshLab's mesh

The next step is to understand how mesh works in MeshLab. MeshLab's defines a CMeshO class. A CMeshO is a collection of vertices and triangular faces. A Face is (mainly) a three pointers structure to vertices that composed it. Every vertex could be referred by zero, one or more faces. A Vertex is (mainly) a class composed by a 3d point in space, a vertex- normal, a color and a 2d texture coordinates.

WARNING!!! *to use vertex-texture coordinates you MUST first enable it, otherwise your plugin will crash!*

You will find all the vertices in an c++ vector called vert and the faces in a vector called face. Typically a MeshLab plugin's function has a MeshModel parameter. A MeshModel contains a CMeshO you could refer to this instance via the cm MeshModel's member. In other words supposing you have a function with a MeshModel parameter called mymesh you could access to CMeshO vertices with:
mymesh.cm.vert[ii]; *//ii is an unsigned int index to a particular vertex in the mesh*

with:
mymesh.cm.face[ii]; *//you can access to the ii face*

and with:
mymesh.cm.face[ii].v(jj); *//you will get the pointer to the jj vertex (0 <= jj < 3) in the ii face*
vcglib also provides a way to simply and safely add and remove faces or vertices from a mesh.

vcg::tri::Allocator<CMeshO>::AddVertex(mymesh.cm,new vertex to be added);
vcg::tri::Allocator<CMeshO>::AddFace(mymesh.cm,new face to be added);

The add functions are defined in the *vcglib\vcg\complex\trimesh\allocate.h* .

To enable texture coordinates per vertex (or for example quality for vertex) you MUST call the function:
mymesh.updateDataMask(MeshModel::MM_VERTTEXCOORD);

to test if the attribute is already enabled:
mymesh.hasDataMask(MeshModel::MM_VERTTEXCOORD);

Please note that you could use also a combination of attributes:
mymesh.updateDataMask(MeshModel::MM_VERTTEXCOORD | MeshModel::MM_FACEQUALITY);
If you need more detailed info about MeshLab/VCGLib mesh you could take a look to this link.

How to write an IO Plugin

Suppose we have to write an importer/exporter for an hypothetical .ext file format. I suggest to put your code in the meshlab's folder *meshlab\src\meshlabplugins\io_ext* . All the subsequent instructions will refer to code inserted in this folder.

MeshLab has a plugin architecture. It means that you have to implement a software interface to let MeshLab's core to interoperate with your plugin. All the softwate interfaces provided by meshlab are in the file *meshlab/src/meshlab/interfaces.h*

You have to develop an input/output plugin. So you have to implement a MeshIOInterface. In other words, supposing you plugin will be called EXTImporter you have to write the something similar to the following code:

```
// file ext_plugin.h
#include "../../meshlab/meshmodel.h" #include "../../meshlab/interfaces.h"
#include "../../meshlab/filterparameter.h" #include<QList>
class EXTImporter : public MeshIOInterface { public: QList<Format>
importFormats(); QList<Format> exportFormats();
  void GetExportMaskCapability(QString &format, int &capability, int
&defaultBits);
  bool open(const QString &format,const QString &fileName,MeshModel &m,int
&mask, const FilterParameterSet & par,vcg::CallBackPos *cb=0,QWidget
*parent=0); bool save(const QString &format,const QString &fileName,
MeshModel &m,const int mask, const FilterParameterSet & par,
vcg::CallBackPos *cb=0,QWidget *parent= 0); };
```

in file ext_plugin.cpp you have to provide implementation to all the previous declared functions.

I will explain the semantics of these functions one by one and what they should do.

- **QList<Format> importFormats()**
- **QList<Format> exportFormats()**
    - they say to MeshLab what kind of extensions the io plugin will read or write.You could copy & paste the following code

```
QList<MeshIOInterface::Format> EXTImporter::exportFormats() const { QList<Format> formatList; formatList << Format("EXT File Format",tr("EXT")); return formatList; }
QList<MeshIOInterface::Format> EXTImporter::importFormats() const { QList<Format> formatList; formatList << Format("EXT File Format",tr("EXT")); return formatList; }
```

- **void GetExportMaskCapability(QString &format, int &capability, int &defaultBits)**
    - it say (mainly) to the framework what kind of extra-attributes (vertex positions must be provided as default) an IO Plugin could save on a file. A typical implementation could be:

```
void EXTImporter::GetExportMaskCapability(QString &format, int &capability, int &defaultBits) const { if(format.toUpper() == tr("EXT")){capability=defaultBits= vcg::tri::io::Mask::IOM_VERTCOLOR | vcg::tri::io::Mask::IOM_FACECOLOR;} }
```

- **bool open(const QString &format,const QString &fileName,MeshModel &m,int &mask,const FilterParameterSet & par,vcg::CallBackPos *cb=0,QWidget *parent=0)**
    - It is THE function you have to implement to import a 3d file format in MeshLab
        - ***Function's parameters:***
            - *const QString &format* - the extension of the format (e.g. "EXT")
            - *const QString &fileName*- the name of the file to be opened
            - *MeshModel &m* - The mesh that is filled with the file content
            - *int &mask* - a bit mask that will be filled reporting what kind of data we have found in the file (per vertex color, texture coords etc). It is a value or a combination of values (via |) of type vcg::tri::io::Mask (you could find it at *vcglib\wrap\io_trimesh\mask_io.h*)
            - *const FilterParameterSet & par* - The parameters that have been set up in the initPreOpenParameter(). IN A FIRST INSTANCE YOU COULD FORGET ABOUT IT.
            - *vcg::CallBackPos *cb* - standard callback for reporting progress in the loading
            - *QWidget *parent* - you should not use this.
- **bool save(const QString &format,const QString &fileName, MeshModel &m,const int mask,const FilterParameterSet & par, vcg::CallBackPos *cb=0,QWidget *parent= 0)**
    - It is the function to save mesh info in a file. it has the same parameters of open function.

How to write a working .pro file

WARNING! *This section will be most an "how to make quickly all the things working" and absolutely not a complete reference to "how write a good pro file"*.
A pro file is a sort of makefile for qt. It is could be usefull in order to compile your plugin directly from your preferred IDE and to create your own meshlab release. In the next steps I will suppose you had put your code in *meshlab\src\meshlabplugins\io_ext*.

- Open with a text editor (also notepad it's ok) the file *meshlab\src\meshlabplugins\meshlabpluginsv12.pro* and add the folder of your own plugin (i.e. io_gsd)
- From the directory *meshlab\src\meshlabplugins\io_ext* take the *io_ext.pro* file.
- copy and paste in *meshlab\src\meshlabplugins\io_ext* then change his name in *io_ext.pro*
- open it with notepad
    - in the TARGET section change the value to io_gsd (this will be the name of your generated plugin's dll)
    - in the HEADERS section remove "io_pdb.h \ $$VCGDIR/wrap/ply/plylib.h" and add all the .h files needed by your plugin
    - in the SOURCES section remove "io_pdb.cpp \ $$VCGDIR//wrap/ply/plylib.cpp\" and add all the .cpp files. Plese, you **MUSTN'T** remove ../../meshlab/filterparameter.cpp otherwise your plugin will have problem on linking time.
- If you use QtCreator or Vs reopen the meshlabv12.pro. You should see your plugin's files listed on the tree view widget.
- Now you can simply compile your plugin!