



# A mesh processing library

Paolo Cignoni

ISTI – CNR

# Intro

- ▣ Intro
- ▣ Capabilities
- ▣ Design/Structure
- ▣ Examples

# What

- A C++ template based library
  - Include only, no compilation hassle
- Research Driven Library
  - The most amatorial professional library
- A rather rich and hopefully easy to use library for mesh processing
- The core of the well known MeshLab system.

# Where

- Main site:
  - <http://vcglib.net>
- The code
  - No rigid release scheme
    - Sync with meshlab releases
  - Just clone the git repo
    - USE the **DEVEL** branch
      - `git clone -b devel https://github.com/cnr-isti-vclab/vcglib.git`
- Documentation by doxygen on the web
- A bunch of small samples
  - `vcglib/apps/sample`

# Capabilities

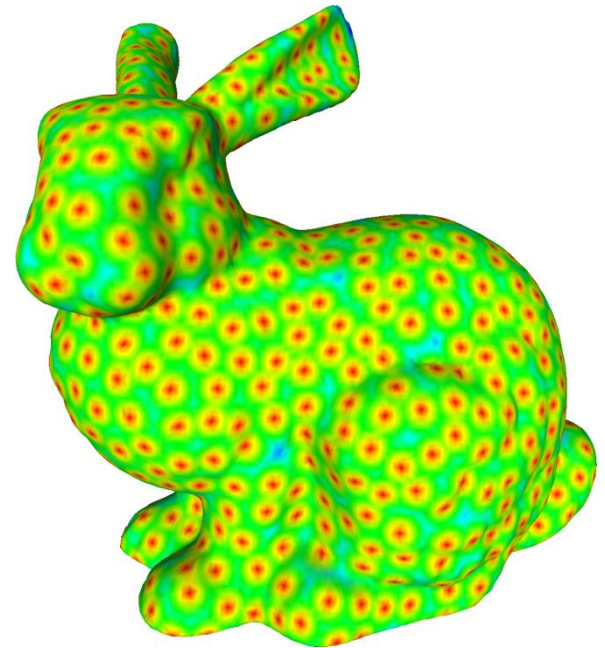
- VCG library feature a large number of different algorithms
- In the next slides a fast browsing of some of the most known things in the library

# Simplification

- Fairly generic edge collapse simplification algorithms
- Probably one of the reason meshlab is famous.
- Link conditions for topology preserving
  - Two optimized specializations
    - Quadric error (with a few minor variants)
    - Quadric error with texture coords optimization.

# Sampling

- A variety of algorithm for distributing points over the surface of a mesh
  - a reasonably practical and fast adaptive poisson sampling algorithm.
  - Unbiased montecarlo
    - Useful for computing sampled integral measures over meshes



# Cleaning

- A variety of tools for correcting small annoying things
  - Duplicated, unreferenced mesh elements
  - Merging of close vertices
  - Small hole filling
  - Non manifold detection and correction
    - Split of non manifold vertexes
    - Heuristic Deletion of isolated non manifold faces



# Color Processing

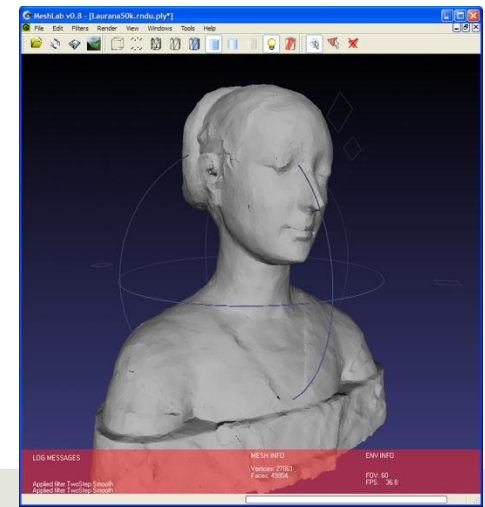
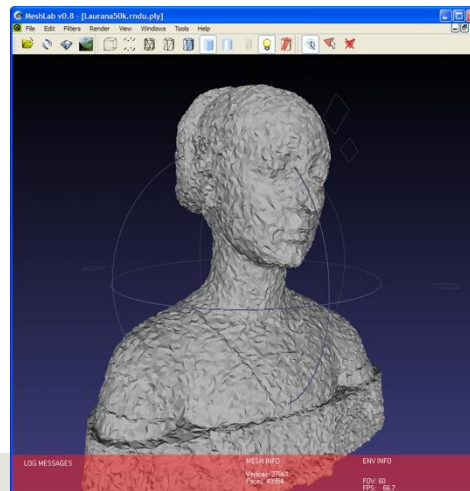
- VCG support color in various format
  - Per vertex
  - Per face
  - Per wedge
  - As texture
- Provides tools for converting from a representation to another one.

# Measuring

- Integral measures
  - Volume, barycenter inertia tensor
- Distance between surfaces
  - Sampled Hausdorff distance
- Distance and intersection between a lot of geometric elements
  - (point-triangle, triangle-triangle etc)

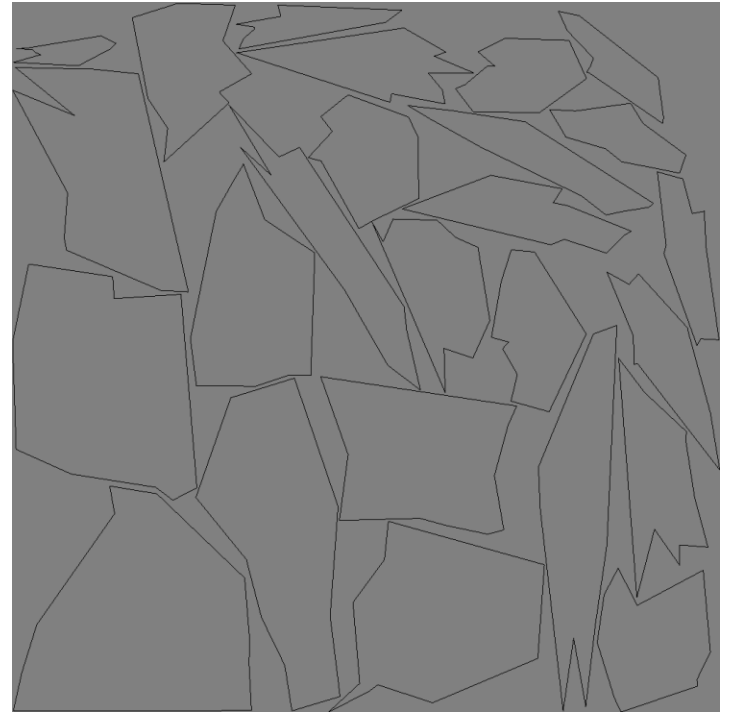
# Smoothing

- A number of sophisticated noise removal tools.
- Basic laplacian (with or without cotangent weighting)
- Taubin smoothing
- Two step feature preserving smoothing.
- A number of smoothing algorithms can also be applied to various attributes like color, normal, scalar field over the mesh



# Texturing

- Support of per vertex and per wedge text coords
- Conversion between representations
- Packing algorithms
- Various texture optimization



# Remeshing

- Subdivision surfaces
  - (loop, butterfly)
  - Generic
    - Define your own predicate to decide if an edge has to be split and where.
- Ball Pivoting surface reconstruction
- Clustering simplification
- Marching cubes

# Spatial Indexing

- Uniform Grid
  - Very good if your query points are quite near to the surface
- Kd-tree
  - Perfect for point clouds
- Hierarchies of Bounding Volumes

# File Format

- 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
- Caveat it flattens everything to a polygon soup.
  - No scene graph information is retained for the most complex formats
  - Many file formats require linking to other piece (only the bold ones are .h pure)

# Basic Concepts: The Mesh

- encode a mesh in several ways,
- the most common is three **vectors** of vertices edges and triangles.
- The following line is an example of the definition of a VCG type of mesh:

```
class MyMesh :  
    public vcg::tri::TriMesh<  
        std::vector<MyVertex>,  
        std::vector<MyFace> ,  
        std::vector<MyEdge> > {};
```

- you need only to derive from `vcg::tri::TriMesh` and to provide the type of containers of the elements



# Basic Concepts: The simplexes 1

- The face, the edge 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 a user defined (possibly empty) collection of attributes
  - For example a vertex could contain position normal color etc.
- To build an simplex class you just derive from the base simplex templated with the desired attributes:

```
class MyVertex2 :
    public vcg::Vertex< MyUsedTypes,
                        vcg::vertex::Coord3f,
                        vcg::vertex::Color4b,
                        vcg::vertex::CurvatureDirf,
                        vcg::vertex::Normal3f,
                        vcg::vertex::BitFlags >{};
```

# Basic Concepts: The simplexes 2

- Caveat first of all you have to pre-declare what are the intended names for the various pieces

```
struct MyUsedTypes : public
    vcg::UsedTypes<
        vcg::Use<MyVertex>      ::AsVertexType,
        vcg::Use<MyEdge>        ::AsEdgeType,
        vcg::Use<MyFace>        ::AsFaceType>{ };
```

- In this way when you are declaring a vertex you already know what are the types involved in mixed relations like the vertex type adjacency

# Basic Concepts: Using the mesh

- Most of the stuff in the library came in the shape of

*static templated class*

- Most of the time you see stuff like

```
vcg::tri::UpdateNormal<MyMesh>::PerVertexNormalized(m);
```

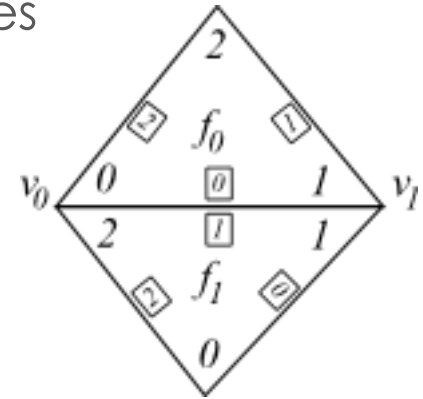
**UpdateNormal** is a static class (no need of an object to be instanced) that act as a container of many algorithms (PerVertexNormalized, PerVertex, PerFaceNormalized, etc...) all devoted to Updating Normals

# Example 1: trimesh\_base

- Basic example of minimal use
- Load a mesh and just dump some info about it
- Note that also the mesh loading is done by mean of templated class.

# Basic Concept: Adjacency

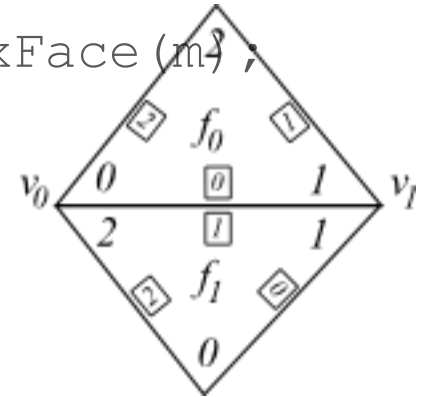
- Vertex, Edge and triangle can store different topological info:
- The most common is the VertexRef field of the face, that store for each triangular face three pointers to its vertexes



- To use certain Adjacency you need the corresponding component in the vertex or face
- FF face face relation
  - `vcg::face::FFAdj`
- VF vertex face relation
  - `vcg::vertex::VFAdj`    `vcg::face::VFAdj`

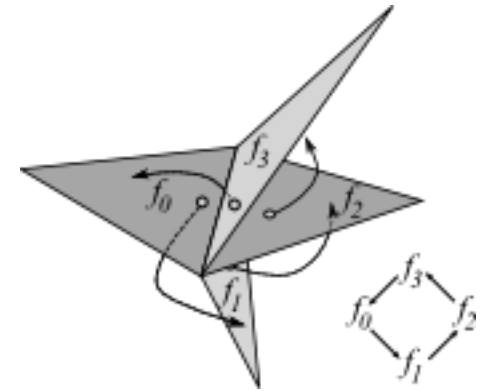
# Basic Concept: Adjacency

- Adjacency Relations must be computed explicitly before being used.
- `tri::UpdateTopology<MyMesh>::FaceFace(m) ;`
- `tri::UpdateTopology<MyMesh>::VertexFace(m) ;`

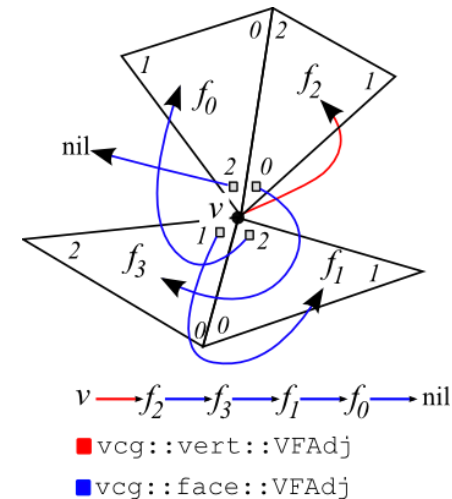


# Basic Concept: Adjacency

- FF relation works for non manifold situations  
faces around an edge are ring connected

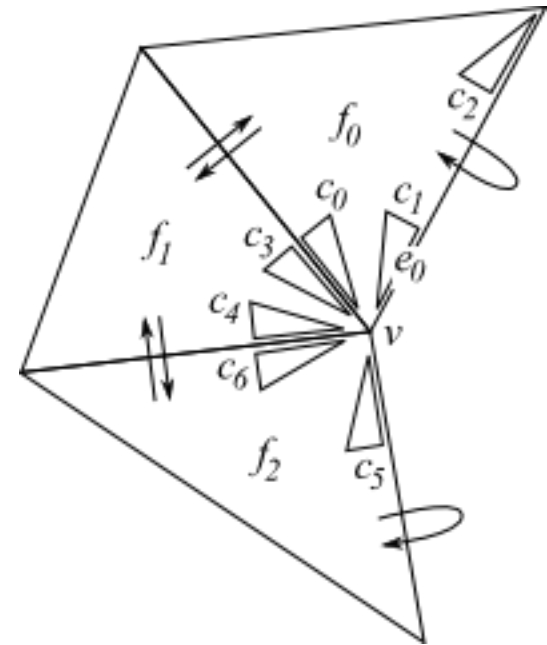


- VF relation does not involve any dynamic allocation,  
the chain of face is distributed onto the involved face



# Basic Concepts: Navigating

- The Pos is the VCG Lib implementation of the Cell-Tuple and it abstracts the concept of position over a mesh
- A Pos in a triangle mesh is a triple made of  
 $\text{pos} = (v, e, f)$
- For manifold meshes there are flip operators that allow easy navigation on the mesh
  - FlipV, FlipE, FlipF
- Each flip operator, applied to a pos simply changes only the indicated element
  - $c2 = c1.\text{FlipV}()$
  - $c0 = c1.\text{FlipE}()$
  - $c3 = c0.\text{FlipF}()$





# Basic Concept: Navigating

- There are also classical retrieval functions:
- `vcg::face::VFOrderedStarFF`
  - Compute the ordered set of faces adjacent to a given vertex using FF adjacency
- `vcg::face::VVStarVF`
- `vcg::face::VFStarVF`
- `vcg::face::VFExtendedStarVF`
- `vcg::face::EFStarFF`

# Example 2: trimesh\_topology

- Note the `face::FFAdj` component in the face
- Note on marking
  - Simplex can have a mark component (`face::Mark`) that offers  $O(1)$  unmark of the whole mesh. Implemented by mean of counters, useful to avoid the usually required  $O(n)$  clearing.
- If your simplex has bitflags, you have also standard visiting/selection bits

# Basic Concept: Allocation

- Simplex are kept into vectors
- Relations are kept by mean of pointers
- Pay attention to reallocations...
  - Always use the library functions to manage the simplex vectors

```
MyMesh::VertexIterator vi = tri::Allocator<MyMesh>::AddVertices(m, 3);  
MyMesh::FaceIterator fi = tri::Allocator<MyMesh>::AddFaces(m, 1);
```

# Basic Concept: De-Allocation

- The library adopts a Lazy Deletion Strategy
  - i.e. the elements in the vector that are deleted are only flagged as deleted, but they are still there.
  - `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
- You can get rid of deleted elements by explicitly calling the two garbage collecting functions:

```
vcg::tri::Allocator<MyMesh>::CompactFaceVector(m);
```

```
vcg::tri::Allocator<MyMesh>::CompactVertexVector(m);
```

# Example 3: trimesh\_allocate

- Note
- How to simply build a minimal mesh from scratch
- the use of the PointerUpdater to cope with vector reallocation
- The use of explicit function to copy a mesh onto another
- The pitfall of having deleted elements

# Basic Concept: 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
  - `tri::Has[attribute](mesh)`
  - `tri::HasPerVertexNormal(m);`
  - `tri::HasPerFaceColor(m);`
  - etc...
- Return a boolean stating if that particular attribute is present or not
- These functions are not statically typed and need the mesh object because of optional stuff...
  - But they are statically solved if no optional stuff arise in your code

# Basic Concept: Requiring data

- Reflection is often used to check the availability of component for a given algorithm
- For example
  - subdivision surface algorithms require FF adjacency
  - Simplification require VF adjacency and per vertex marks
  - Etc.
- If something is missing an exception is raised
- `Tri::RequireFFAdjacency(mesh);`
  - Raise a **missing component** exception if the FF adj is missing

# Basic Concept: Optional Component

- Simplex components imply storage
  - E.g. FF adjacency means 4 words per face.
  - Components are stored into the simplex type
- Most components can be done optional
  - E.g. you can control the allocation space of that component at runtime

```
class CFaceOcf      : public vcg::Face< MyUsedTypesOcf,  
    vcg::face::InfoOcf, vcg::face::FFAdjOcf,  
    vcg::face::VertexRef, vcg::face::BitFlags,  
    vcg::face::Normal3fOcf > {};
```

```
class CMeshOcf      : public vcg::tri::TriMesh<  
    vcg::vertex::vector<CVertex>,  
    vcg::face::vector_ocf<CFaceOcf> > {};
```



# Basic Concept: Optional Component

- Storage of optional component is separated
  - E.g. The data for the FF adjacency is stored in a 'parallel' vector alongside the face vector.
- Access is exactly the same.
- You explicitly control the allocation

```
assert (tri::HasFFAdjacency (cmof) == false);  
cmof.face.EnableFFAdjacency ();  
assert (tri::HasFFAdjacency (cmof) == true);
```

# Example4: trimesh\_optional

- Note the different definition of the type
- Note the enabling of the needed components
- Try to raise exceptions by commenting out the needed enabling

# Basic Concept: User Def Attribute

- VCG Lib provides a mechanism to associate **user-defined 'attributes'** to the simplicies and to the mesh
- Attribute vs Components
  - Components are conceptually inside the simplex
    - `(*vi).N();`
  - Attributes need an handle to be accessed
    - `irradHandle[vi];`
- To use an attribute
  - Build an handle (find or create the attribute)
  - Use the handle to access the data

# Basic Concept: User Def Attribute

## ■ Getting a named attribute handle

```
MyMesh::PerVertexAttributeHandle<float> named_hv =  
    vcg::tri::Allocator<MyMesh>::GetPerVertexAttribute<float>  
        (m, std::string("Irradiance"));
```

## ■ Using an handle

```
MyMesh::VertexIterator vi; int i = 0;  
for(vi = m.vert.begin(); vi != m.vert.end(); ++vi, ++i)  
{  
    named_hv[vi]    = 1.0f; // [] operator takes a iterator  
    named_hv[*vi]  = 1.0f; // or a MyMesh::VertexType object  
    named_hv[&*vi]= 1.0f; // or a pointer to it  
    named_hv[i]    = 1.0f; // or an integer index  
}
```

# Basic Concept: *ForEach* construct

- to traverse all the vertexes of a mesh you can simply write something like:

```
ForEachVertex(m, [&](const VertexType &v) {  
    MakeSomethingWithVertex(v);  
});
```

- There are similar constructs for edges and faces
- Main advantage, avoid verbose checking of deleted elements

```
MyMesh::VertexIterator vi;  
for(vi = m.vert.begin(); vi != m.vert.end(); ++vi, ++i)  
{  
    if(!vi->IsD())  
    {  
        MakeSomethingWithVertex(v);  
    }  
}
```

# Example5: trimesh\_attribute

- Note the creation/test/delete functions
- Note the multiple way of accessing thru handles