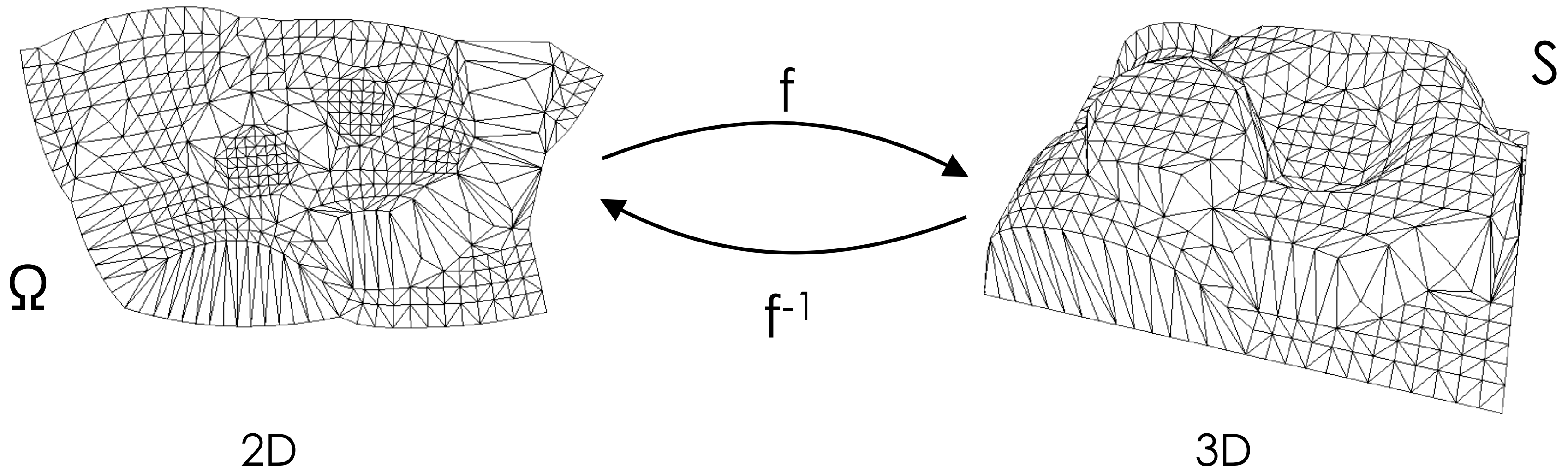
FGT: 2015/2016
Parametrization & Remeshing

Nico Pietroni (nico.pietroni@isti.cnr.it)

ISTI – CNR

Recap

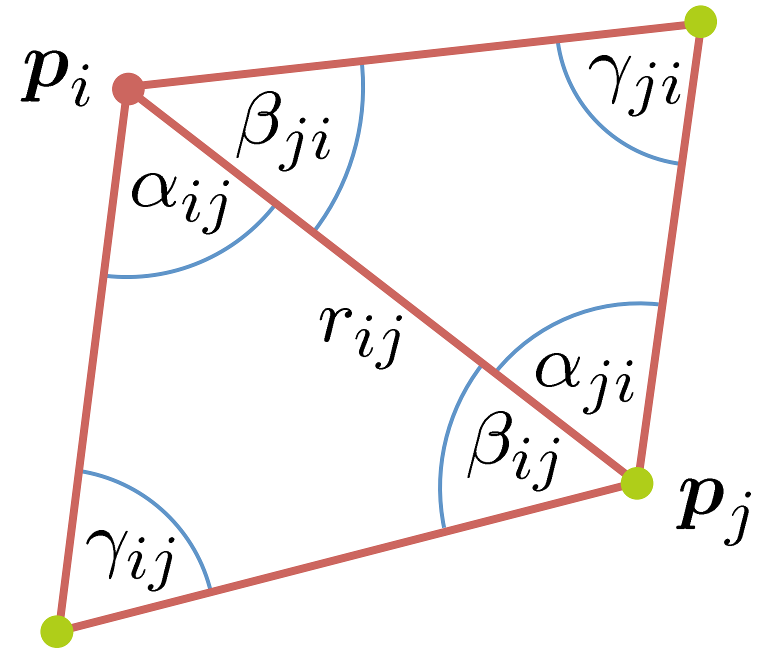
- Parametrization



Weight ed average

- discrete harmonic coordinates

$$w_{ij} = \cot \gamma_{ij} + \cot \gamma_{ji}$$

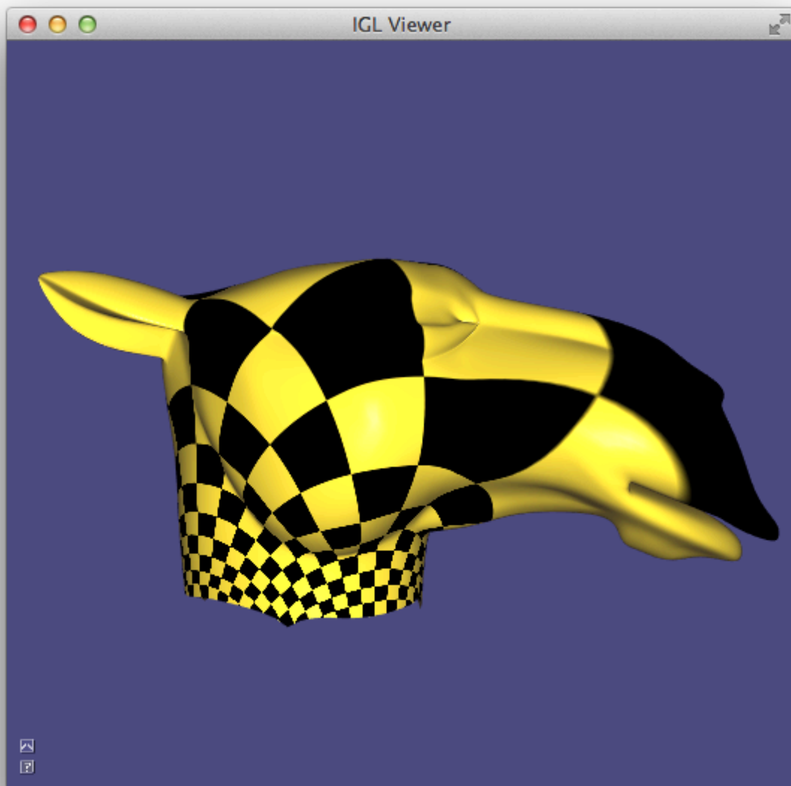


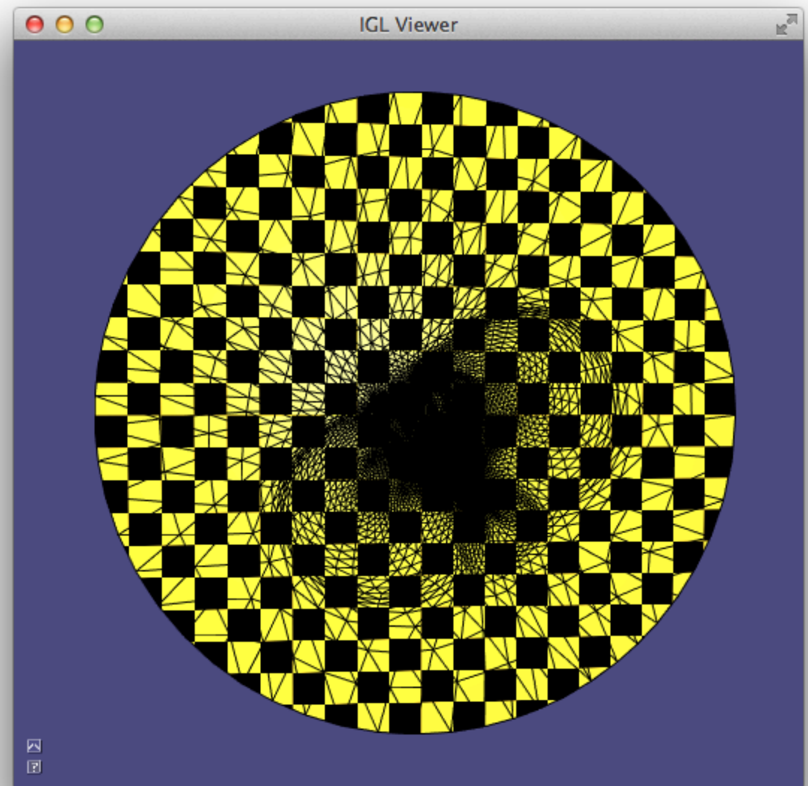
normalization

$$\lambda_{ij} = \frac{w_{ij}}{\sum_{k \in N_i} w_{ik}}$$

Harmonic parametrization

- Fix the boundary of the mesh to UV
- Express each UV position as linear combination of neighbors
- Use cotangent weights!





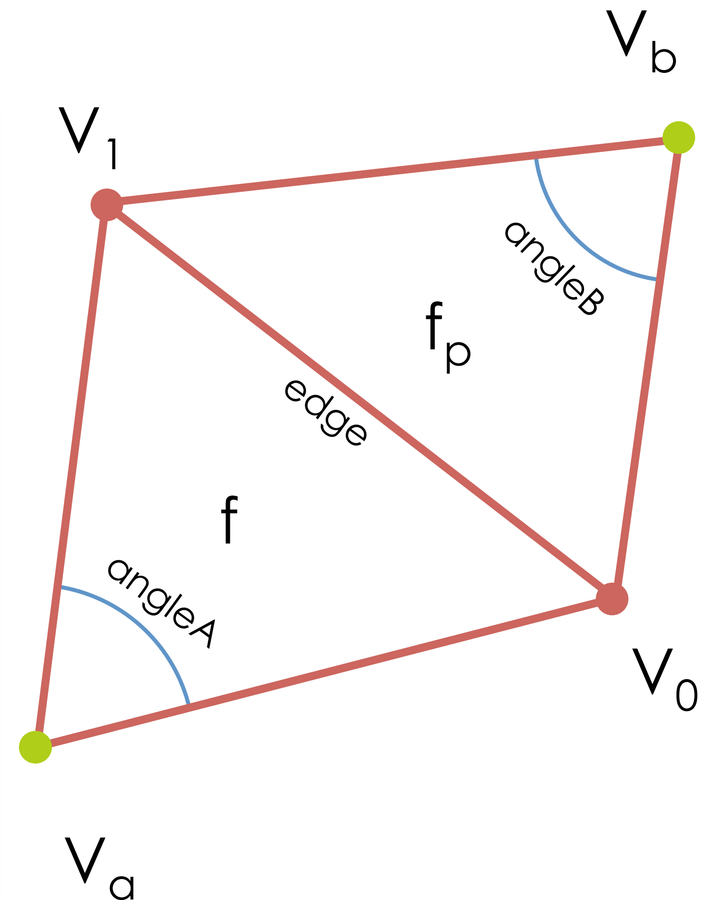
Implementation H-Maps (0)

- Linear sistem
- Sparse matrix (2n x 2n), where n is number of vertices of the mesh
- Express each point as weighted sum of its neighbors
- Find x such that Ax=0
- x are the final UV coordinates!

Implementation H-Maps (1)

Computing cotangent matrix

```
template <typename ScalarT>
    static ScalarT CotangentWeight(const FaceType &f, int edge)
    {
        const FaceType * fp = f.cFFp(edge);
        assert(edge >= 0 && edge < 3);

        ScalarT cotA = 0;
        ScalarT cotB = 0;

        // Get the edge (a pair of vertices)
        VertexType * v0 = f.cV(edge);
        VertexType * v1 = f.cV((edge+1)%f.VN());

        if (fp != NULL &&
            fp != &f)
        {
            // not a border edge
            VertexType * vb = fp->cV((f.cFFi(edge)+2)%fp->VN());
            ScalarT angleB = ComputeAngle<ScalarT>(v0, vb, v1);
            cotB = vcg::math::Cos(angleB) / vcg::math::Sin(angleB);
        }

        VertexType * va = f.cV((edge+2)%f.VN());
        ScalarT angleA = ComputeAngle<ScalarT>(v0, va, v1);
        cotA = vcg::math::Cos(angleA) / vcg::math::Sin(angleA);

        return (cotA + cotB) / 2;
    }
```



Implementation H-Maps (2)

Cycle over all faces, cumulate per face/edge contributes

```
vcg::tri::UpdateFlags<MeshType>::FaceClearV(m);
for (size_t i = 0; i < m.face.size(); ++i)
{
    FaceType & f = m.face[i];
    assert(!f.IsV());
    f.SetV();

    // Generate coefficients for each edge
    for (int edge = 0; edge < 3; ++edge)
    {
        CoeffScalar weight;
        const FaceType * fp = f.cFFp(edge);
        if ((fp == NULL) ||  (fp == &f)) continue;
        if (fp->IsV()) continue;

        weight = CotangentWeight<ScalarT>(f, edge);

        size_t v0_idx = vcg::tri::Index(m, f.V0(edge));
        size_t v1_idx = vcg::tri::Index(m, f.V1(edge));

        coeffs.push_back(Eigen::Triplet<CoeffScalar>(v0_idx, v1_idx, -weight));
        coeffs.push_back(Eigen::Triplet<CoeffScalar>(v1_idx, v0_idx, -weight));

       }
}

// Setup the system matrix
Eigen::SparseMatrix<CoeffScalar> laplaceMat;        // the system to be solved
laplaceMat.resize(n, n); // eigen initializes it to zero
laplaceMat.reserve(coeffs.size());
laplaceMat.setFromTriplets(coeffs.begin(), coeffs.end());
```

Implementation H-Maps (3)

- Normalize
  - Add sum on diagonal

```
for (size_t i = 0; i < m.face.size(); ++i)
{
    ….
    for (int edge = 0; edge < 3; ++edge)
    {
       …
       sums[v0_idx] += weight;
       sums[v1_idx] += weight;
    }
}

for (std::map<size_t,CoeffScalar>::const_iterator it = sums.begin(); it != sums.end(); ++it)
    coeffs.push_back(Triple(it->first, it->first, it->second));
```

Implementation H-Maps (4)

Boundary Constraints on vertices

- **Use Lagrange Multipliers**
  - Suppose I've to solve Ax=f and A is 3x3 matrix
  - subject to $x_1 + x_2 = a$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & 0 \\ a_{10} & a_{11} & a_{12} & 1 \\ a_{20} & a_{21} & a_{22} & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \lambda_0 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ a \end{bmatrix}$$

- Implemented on **LibIGL**

Implementation H-Maps (5)

Boundary Constraints on vertices

- **Use Penalization factors**
  - Suppose I've to solve Ax=f and A is 3x3 matrix
  - subject to x_1=a
  - Take a very hing K

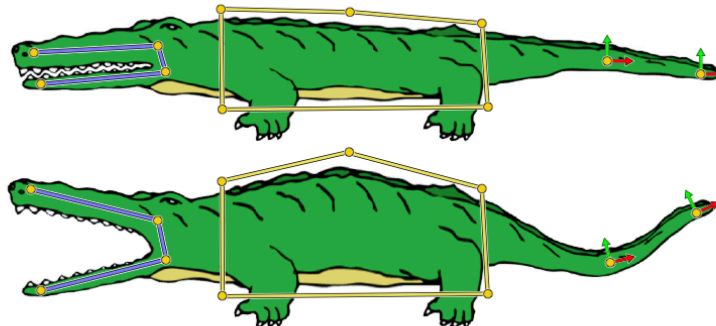
$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11}+K & a_{12} \\ a_{20} & a_{20} & a_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ Kb \\ f_2 \end{bmatrix}$$

- Implemented on vcg harmonic.h

Harmonic Weights

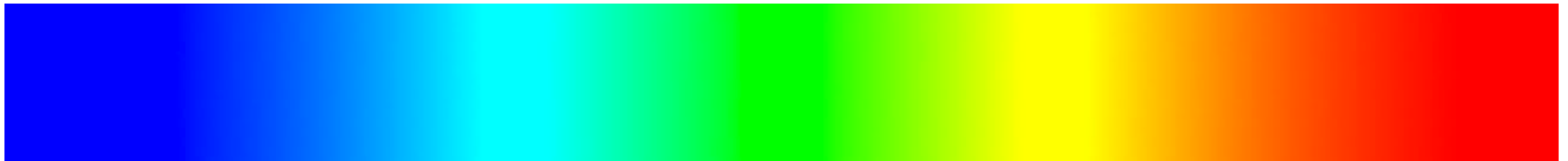
- Used to smoothly interpolate scalar values over a mesh given some sparse constraint

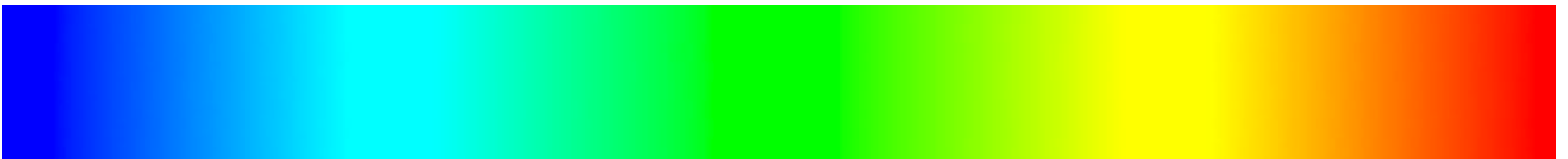
- Useful to interpolate deformations



Bi-Harmonic

- Harmonic minimize the variance of 1° derivative
- Bi-harmonic minimize the variance of 2° derivative
  - Bi-harmonic is simply obtained multiplying cot matrix by himself!



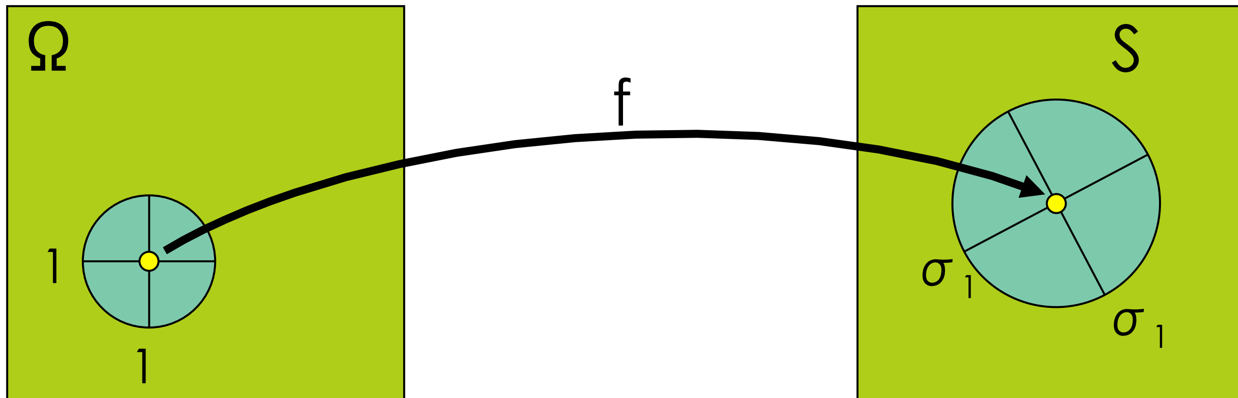


- Overshooting problem with Biharmonic

Bounded Bi-Harmonic

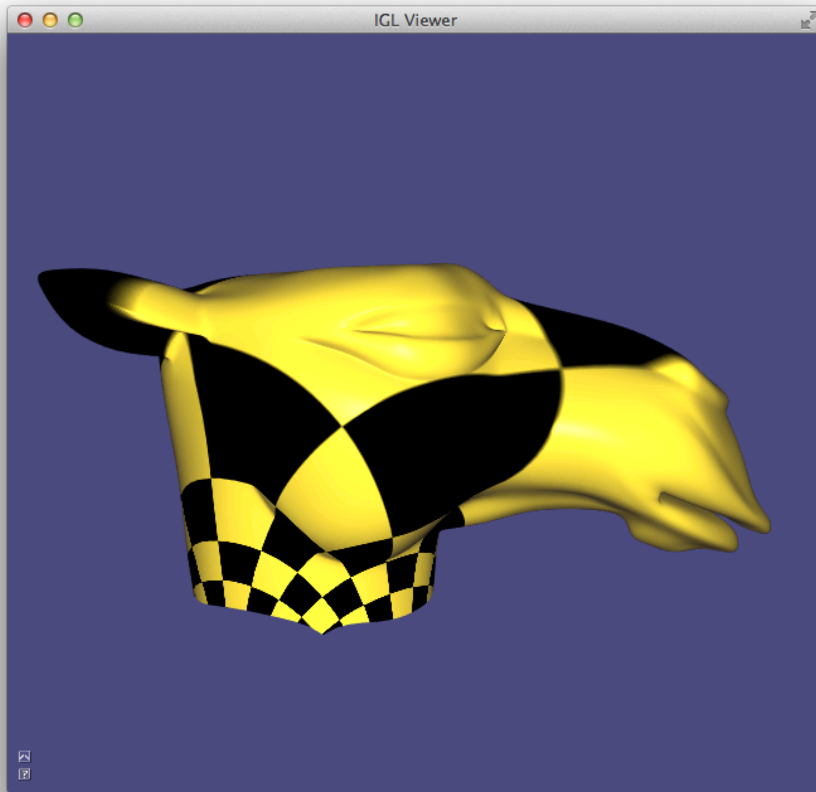
Least Squares Conformal maps

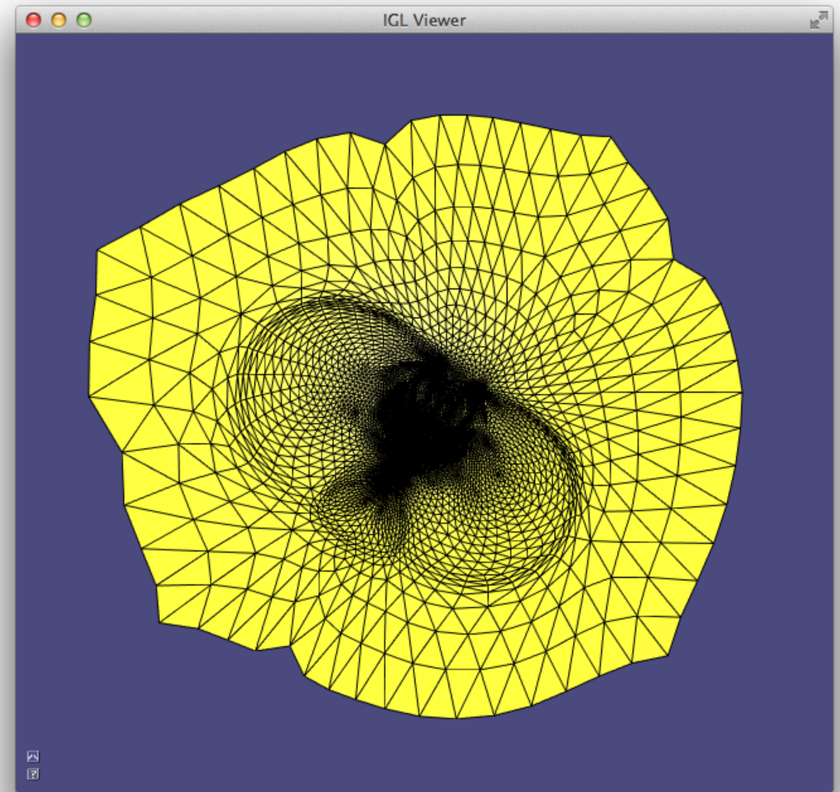
- Doesn't need the entire boundary to be fixed
- Imposing that two vectors un UV maps to 2 orthogonal, same length vectors in 3D.



Least Squares Conformal maps

- Need to fix only 2 vertices to disambiguate
- Why?





LSCM Parametrization (VCG)

Least Squares Conformal Maps

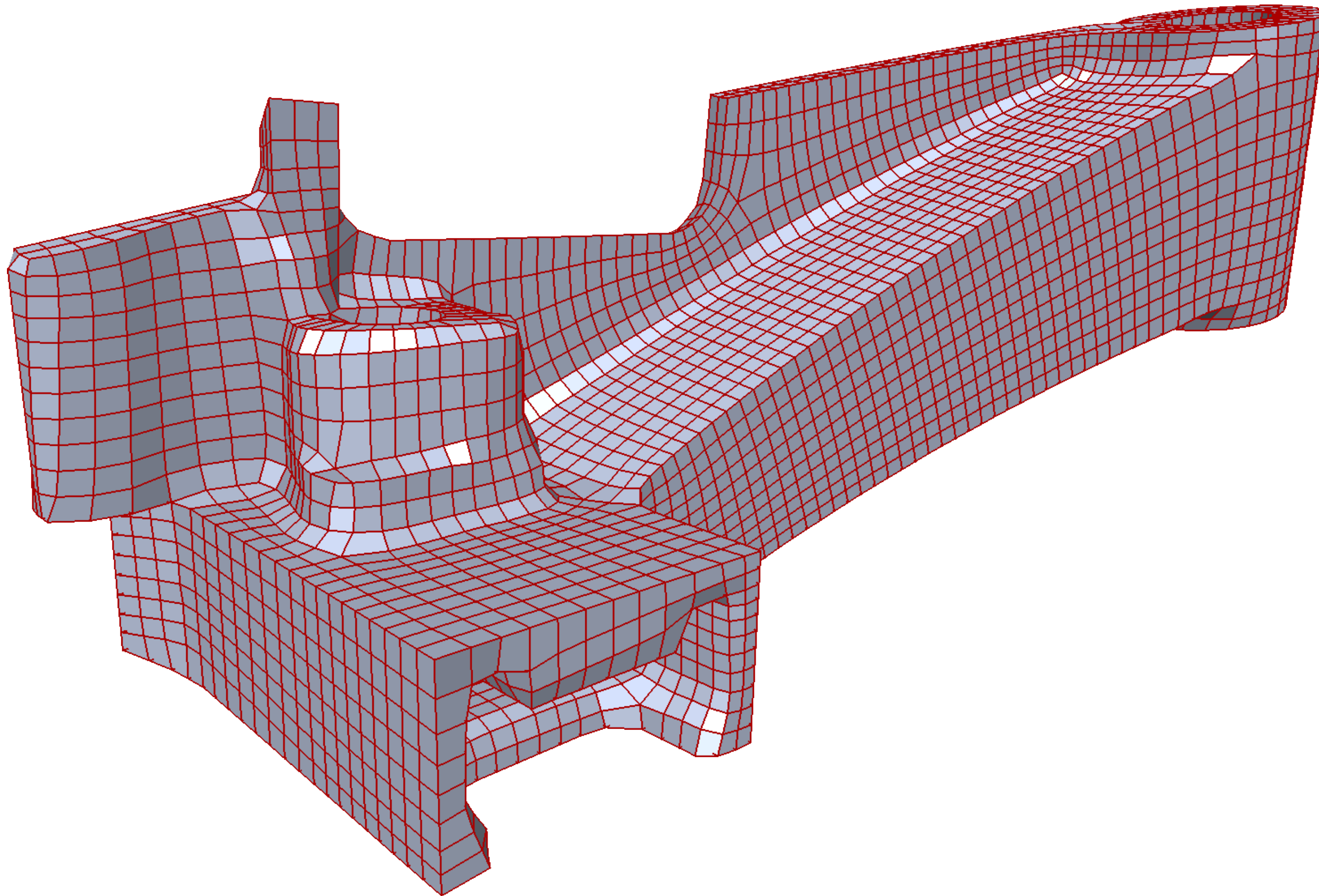
```
#include <vcg/complex/algorithms/parametrization/poisson_solver.h>

MyTriMesh m;
tri::PoissonSolver<MyTriMesh > PS(m);

if(!PS.IsFeaseable())
{
  printf("mesh is not suitable for parametrization \n");
  return -1;
 } else
 printf("OK - mesh is homeomorphic to a disk\n");

PS.Init();
PS.FixDefaultVertices();
PS.SolvePoisson(true);
```

Possibile to choose vertices to fix

```
PS. SetSelectedAsFixed();
PS. SetBorderAsFixed(true);
```

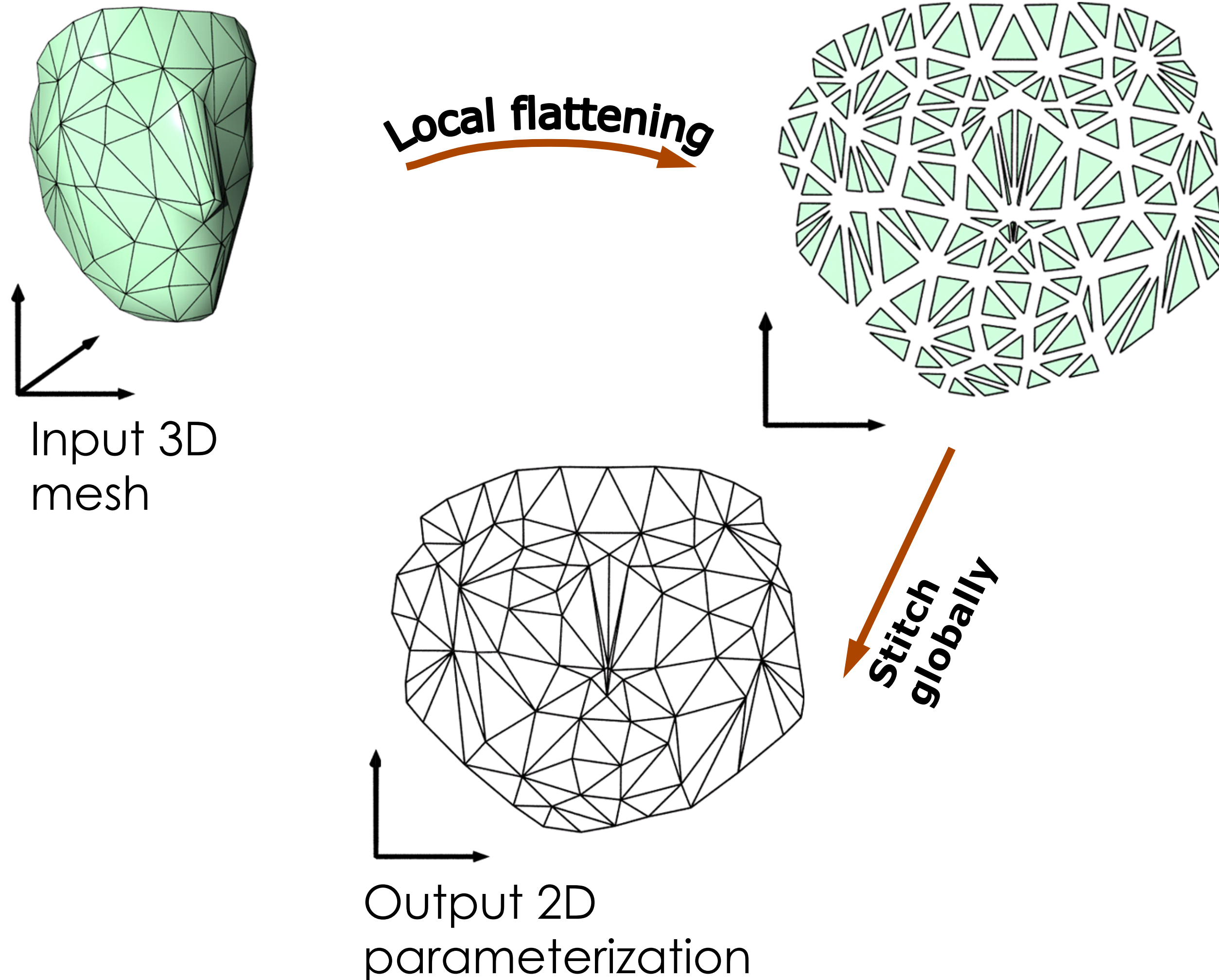
LSCM field Aligned

Modified to align the gradients to a given field



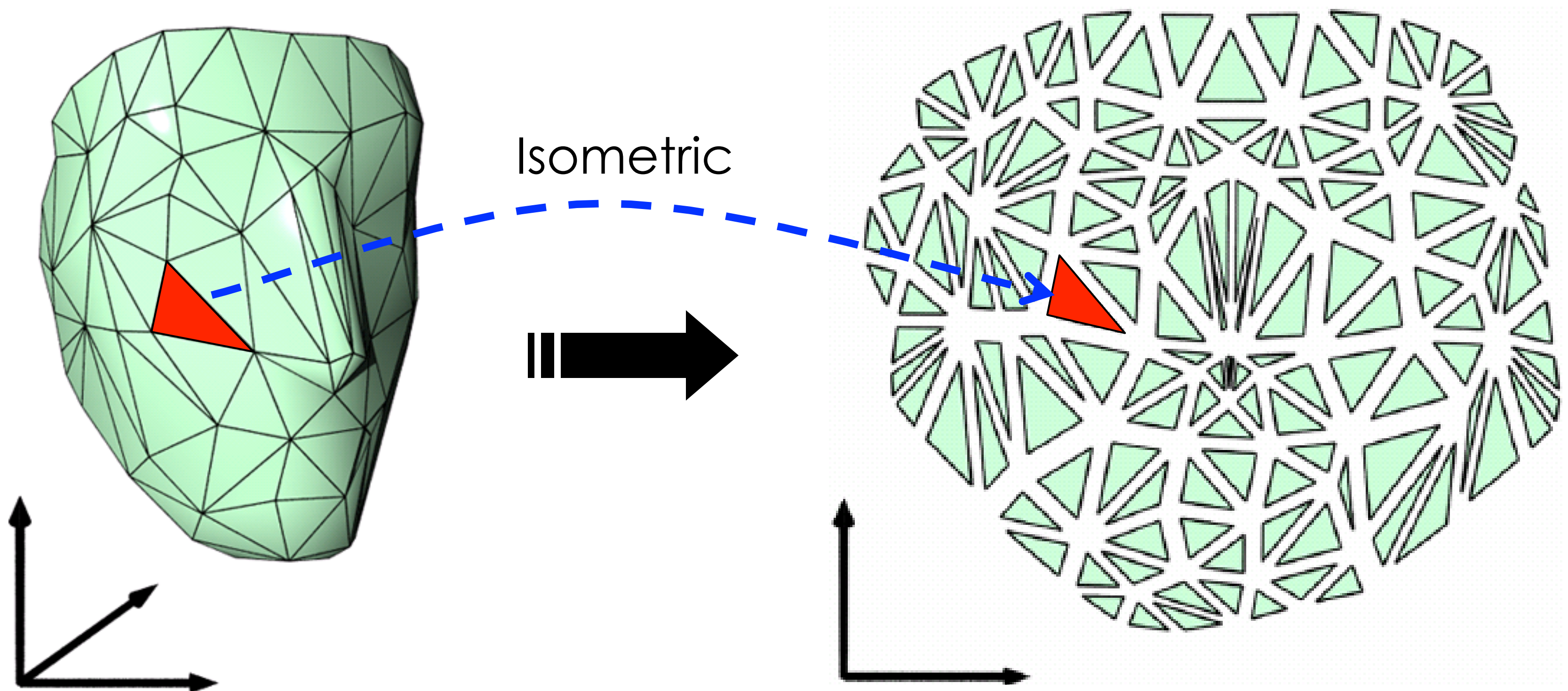
As-rigid-as-possible parametrization (0)

Local-Global Approach



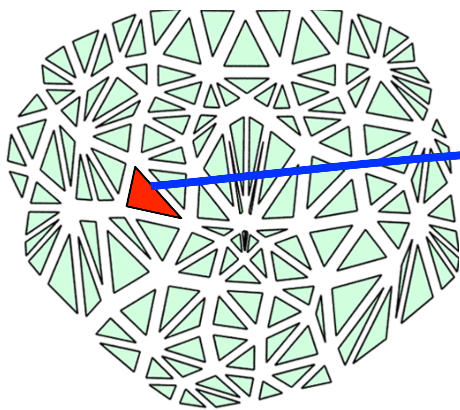
As-rigid-as-possible parametrization (1)

- Each individual triangle is independently flattened into plane without any distortion

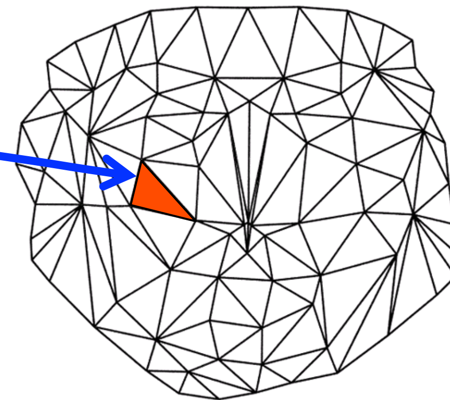


As-rigid-as-possible parametrization (1)

- Merge in UV space (averaging or more sophisticated strategied)



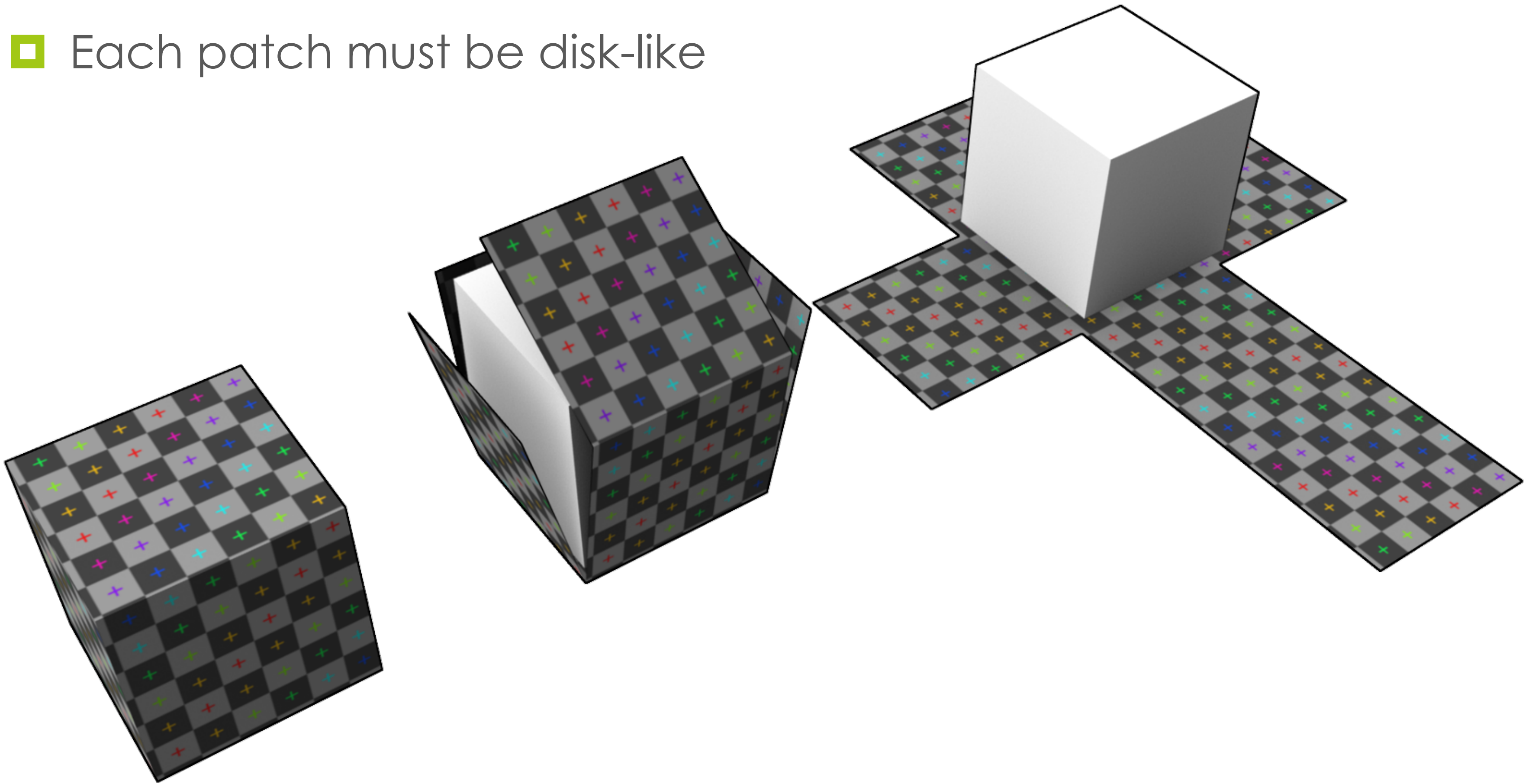
Reference triangles x



Parameterization *u*

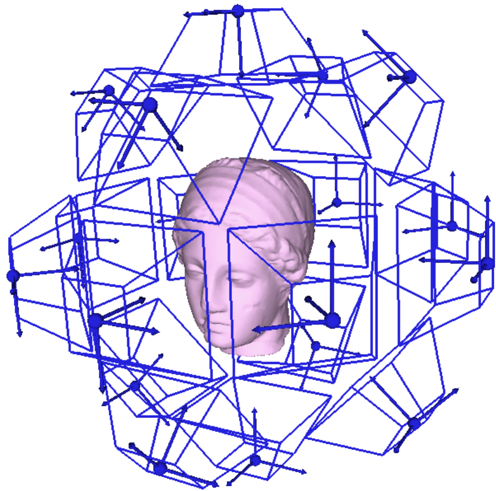
Deriving Cuts

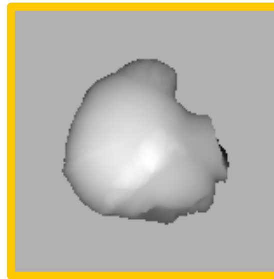
- Splitting the mesh in sub-partitions
- Each patch must be disk-like

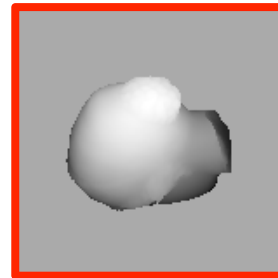


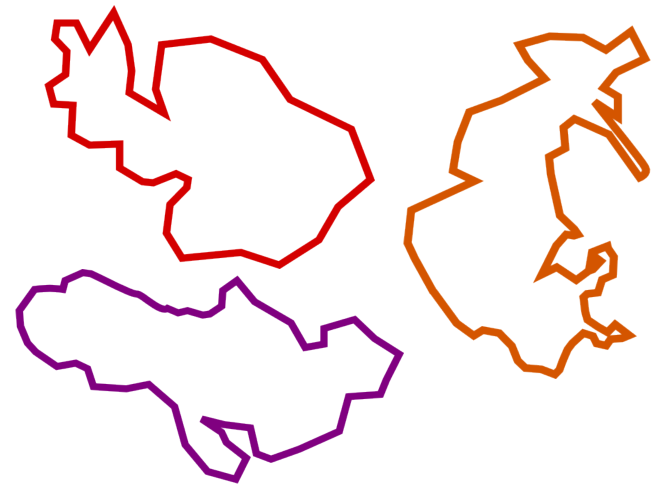
Orthoprojection (0)

- Use orthographics Projection
- Map each triangle in the “best projection”
- Use of depth peeling



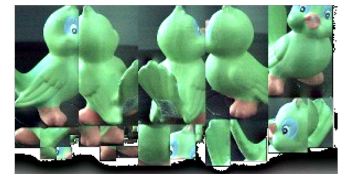






Orthoprojection (1)

- Isolated pieces are removed and merged with bigger areas, to avoid fragmentation
- Useful for Color-to-Geometry mapping

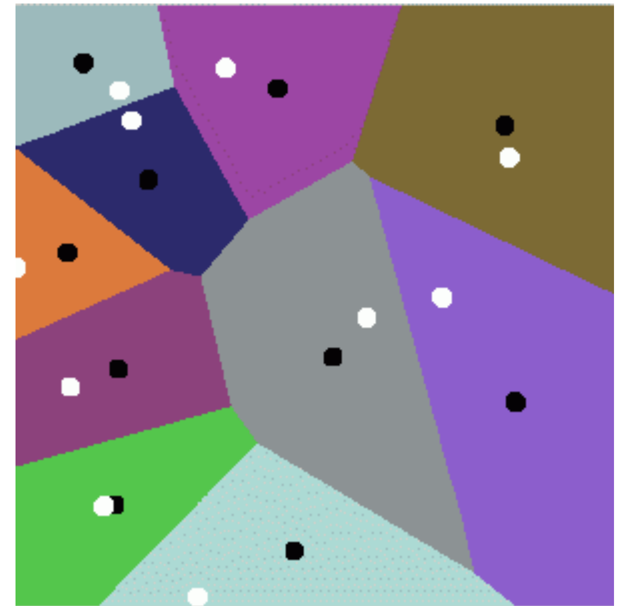


Centroidal Voronoi Diagrams

- VD generated from random seeds are not well placed around the seeds.
- Region are not “centered” around the seeds

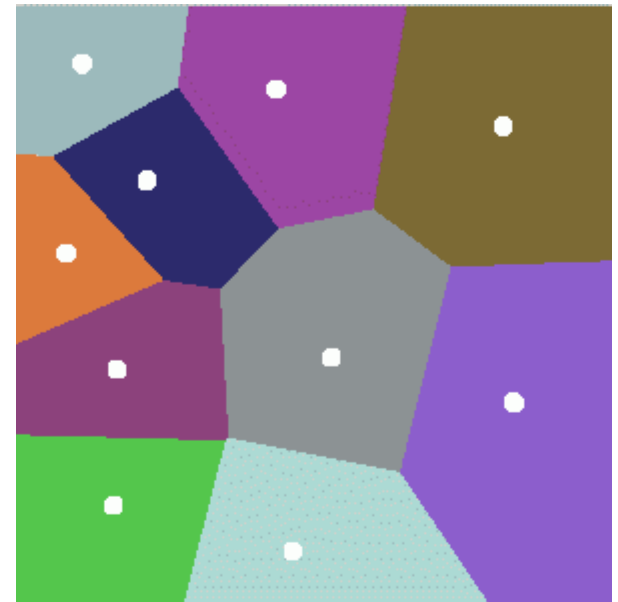
Centroidal Voronoi Diagrams

- VD generated from random seeds are not well placed around the seeds.
- Region are not "centered" around the seeds



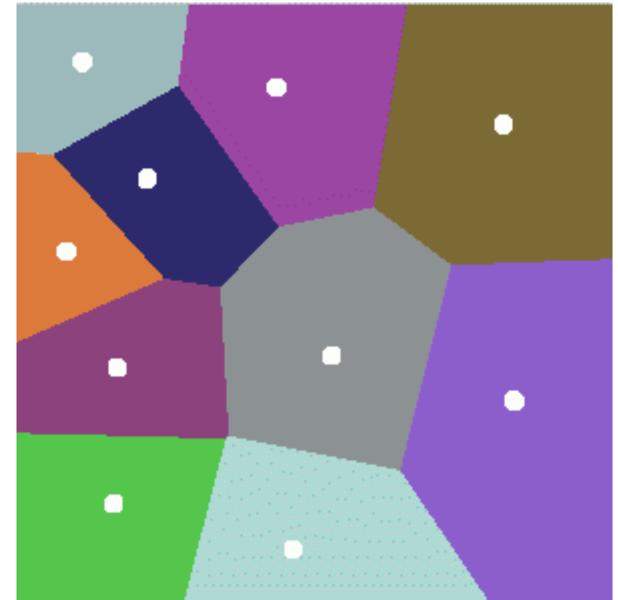
Centroidal Voronoi Diagrams

- VD generated from random seeds are not well placed around the seeds.
- Move the seeds toward the centroid of the region
- Recompute VD



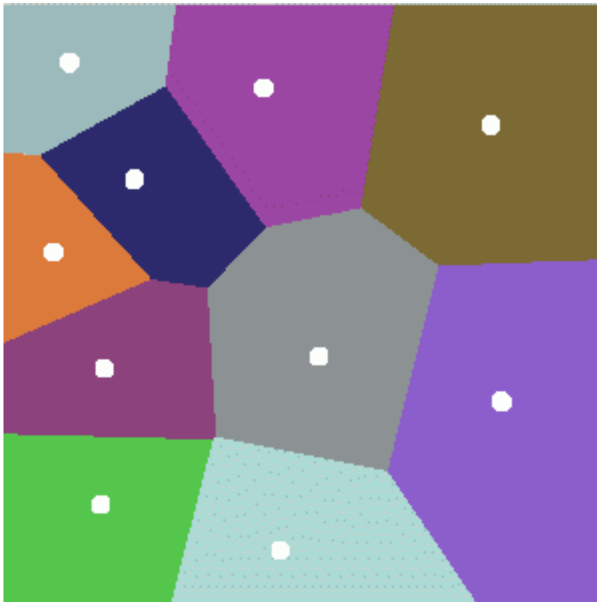
Centroidal Voronoi Diagrams

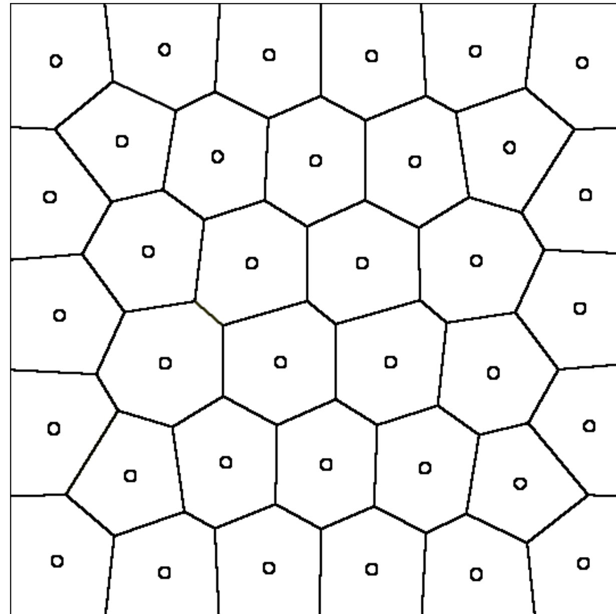
- Lloyd’s Relaxation
- Until VD sites are not centroids
  - move site to centroid,
  - recalculate VD



Voronoi Partitioning (0)

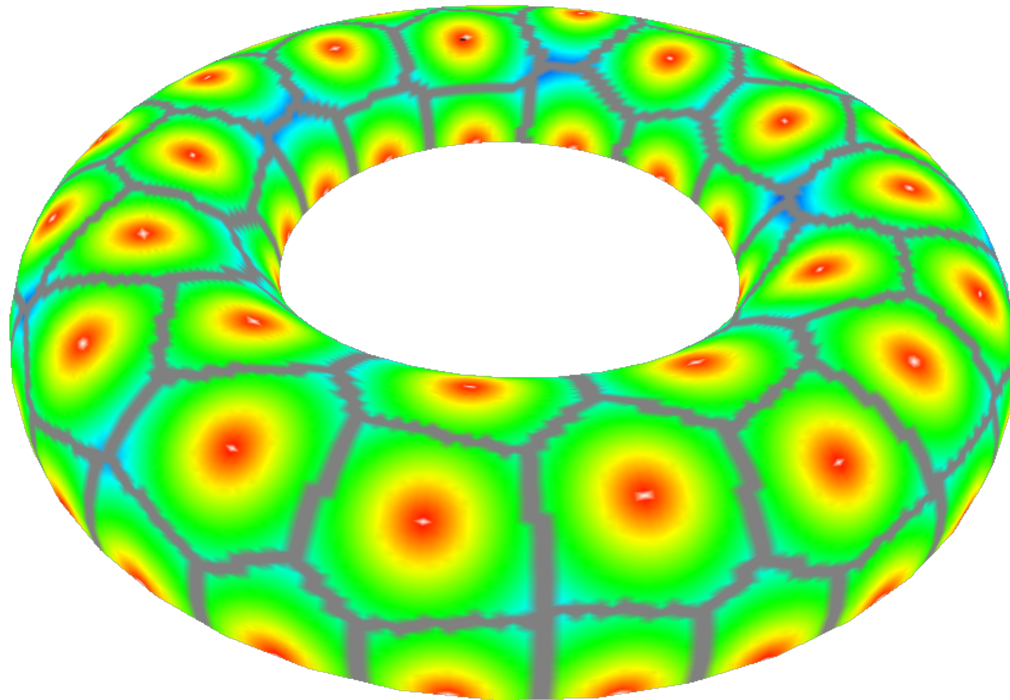
- Use voronoi centroidal relaxation
  - Generate random
  - Move the seeds toward the centroid of the region
  - Recompute VD
  - https://www.jasondavies.com/lloyd/





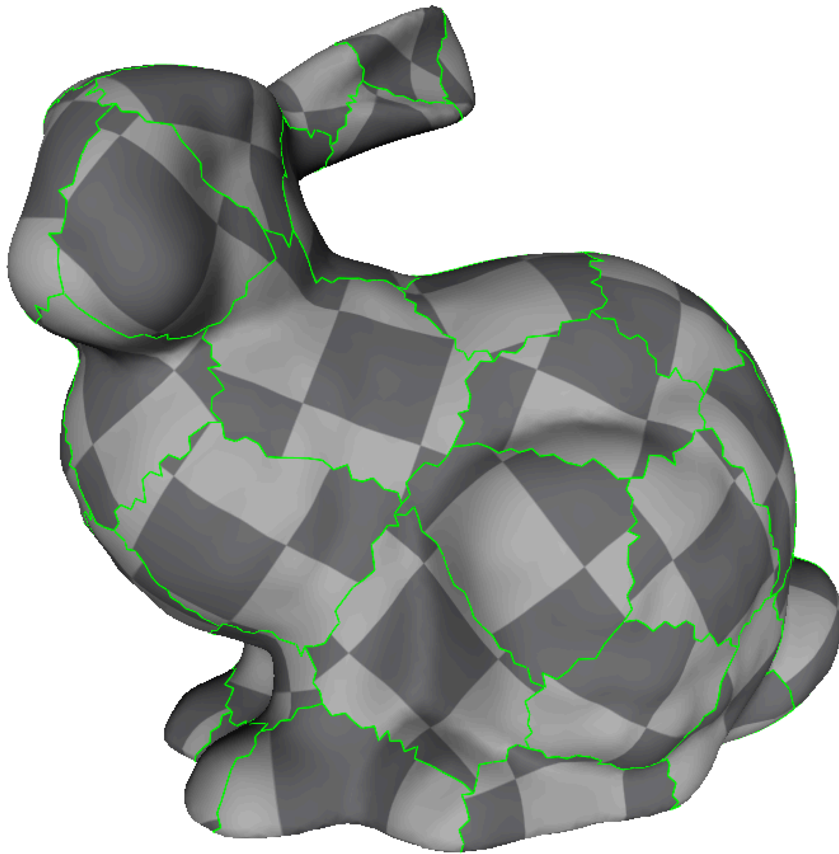
Voronoi Partitioning (1)

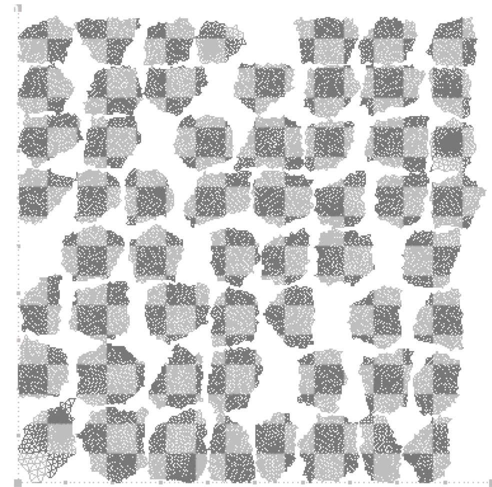
- On mesh's surface give a nice patch layout!



Voronoi Partitioning (2)

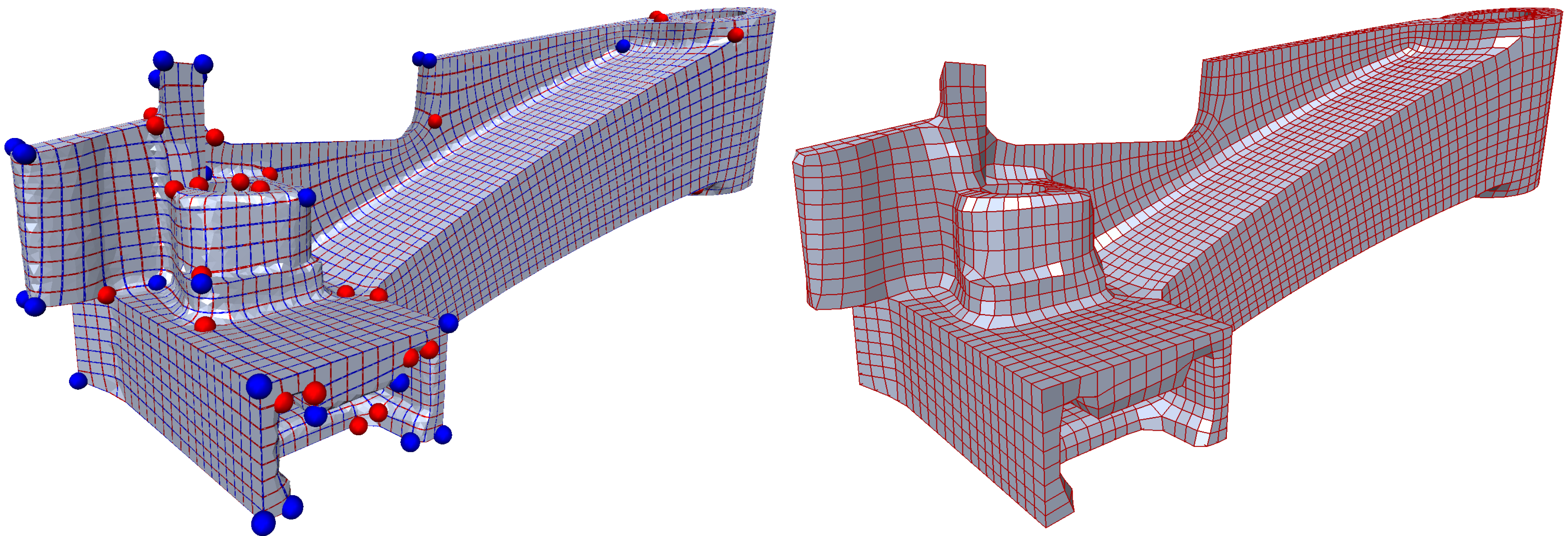
- That can be used for parametrization





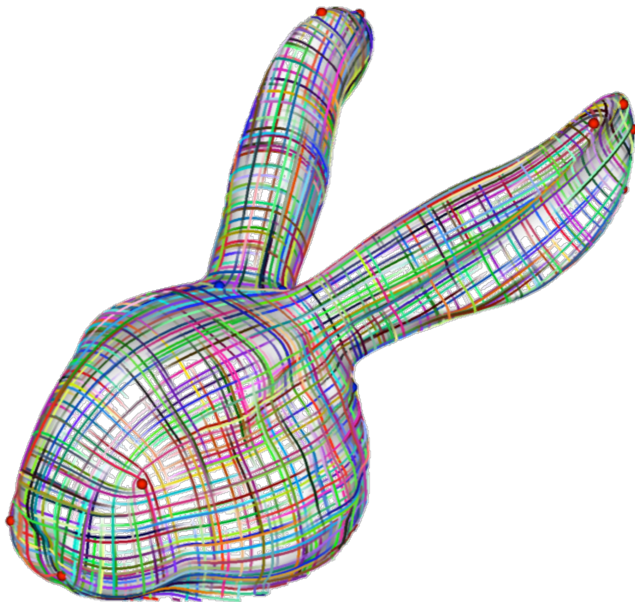
Parametrization for remeshing: field alignment(0)

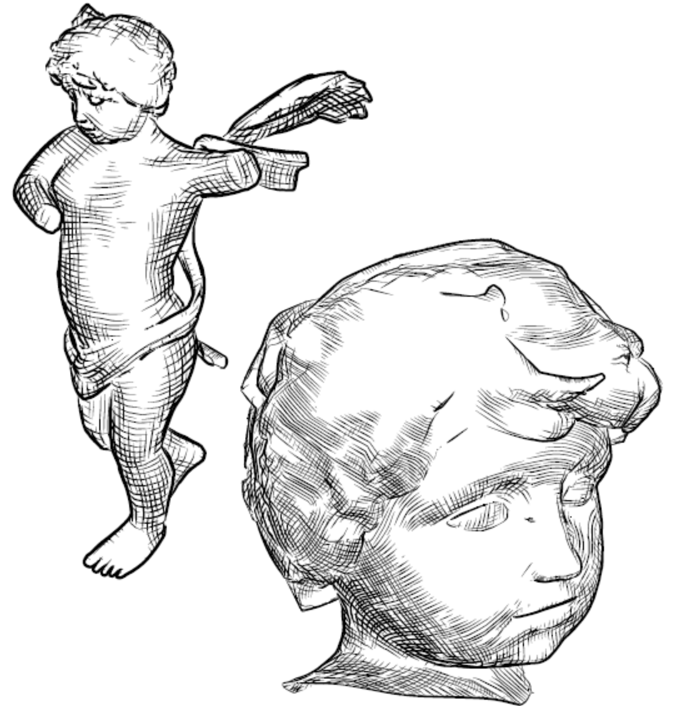
- Crucial to place singularities in the right place!



Parametrization for remeshing: field alignment(1)

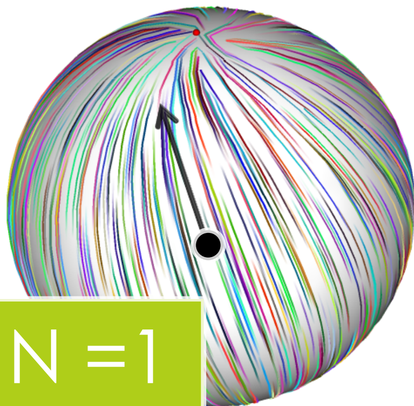
- Usually aligned to a smooth cross field
- Which follows main curvature directions
- And is aligned to the main features

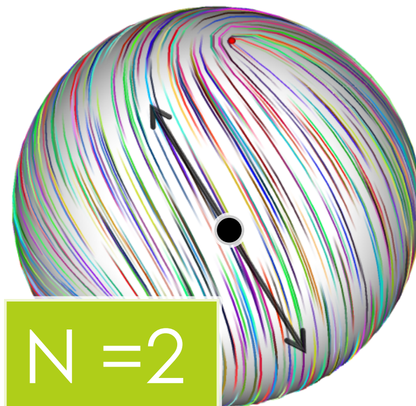


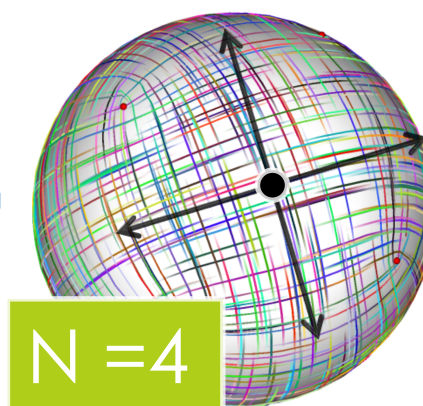


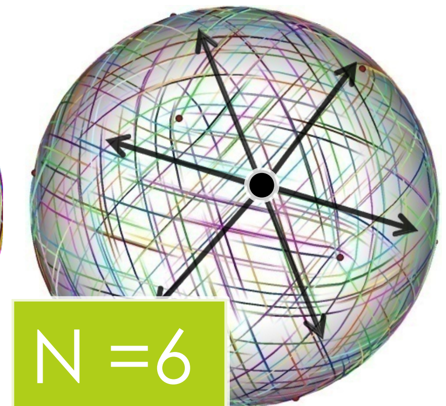
What is a Field?

- A direction field is, for each point of a surface, a set of N unit vectors of the tangent plane that is invariant by rotation of $2\pi/N$.



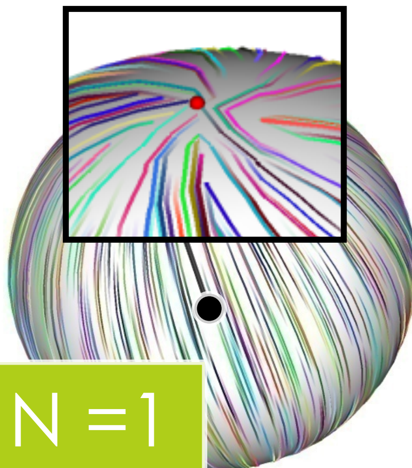


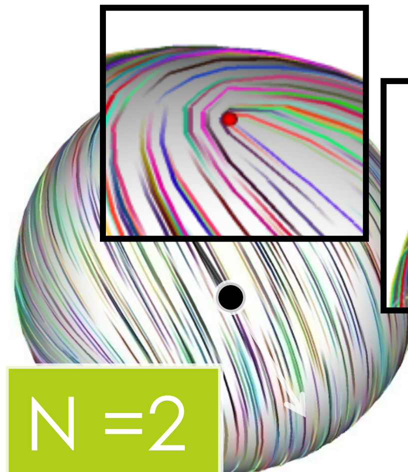


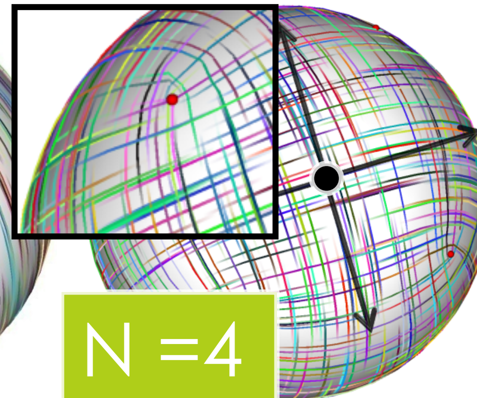


What is a Singularity?

- Singularities are a generalization of the poles (and saddles) of vector field.

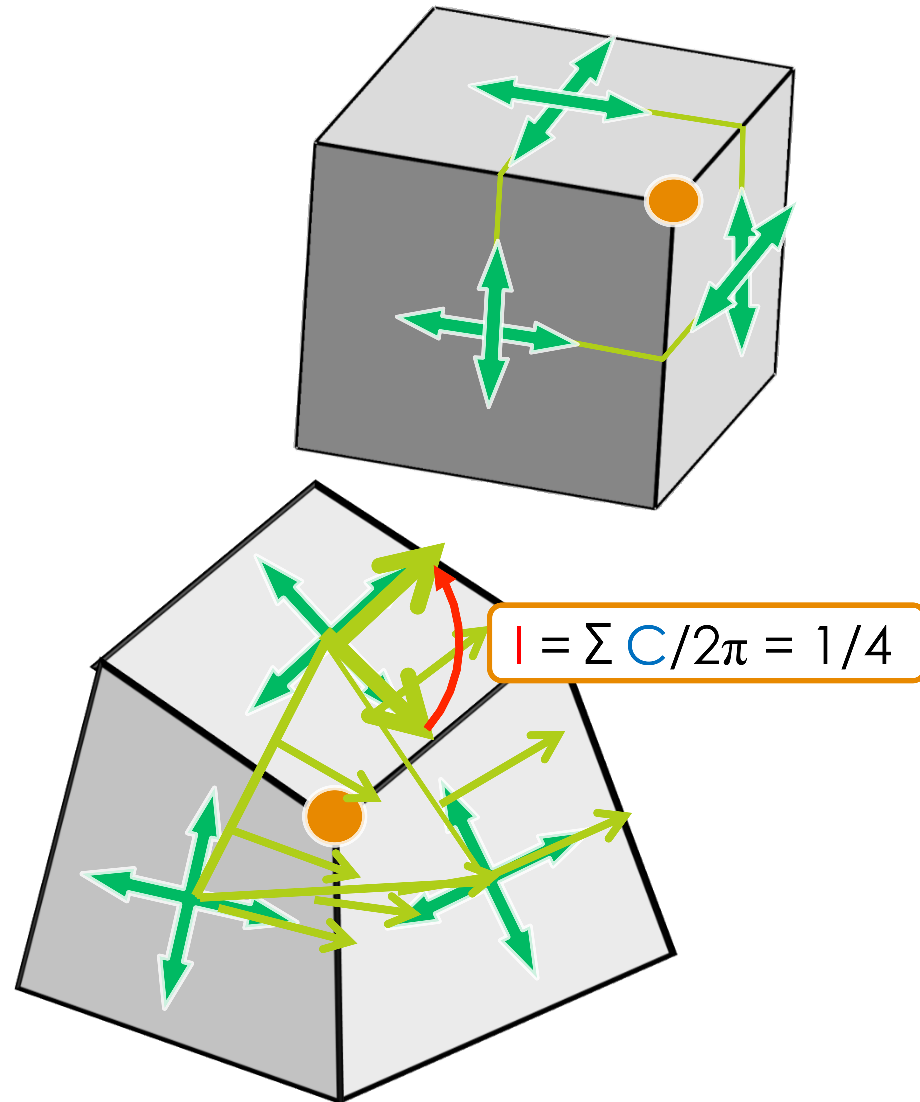






Cross Field: Discrete settings

- We rely on N=4 for quad remeshing
- In the discrete settings
- One cross field per face
- Evaluate singularities in a local map



Cross Field(VCG)

Both per Face and per Vertex

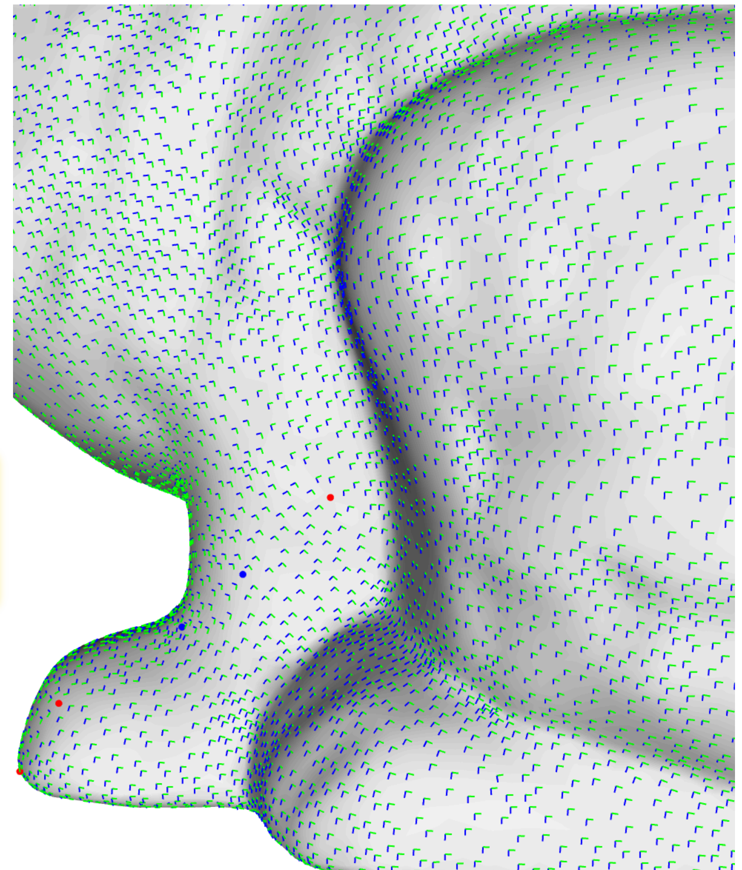
```
class MyTriVertex:public vcg::Vertex<TriUsedTypes...,vcg::vertex::CurvatureDird, ... >{};

class MyTriFace:public vcg::Face<TriUsedTypes...,vcg::face::CurvatureDird,... >{};
```

Access

```
MyTriVertex *v= ...;
Vcg::Point3d Dir1=v->PD1();
Vcg::Point3d Dir2=v->PD2();

MyTriFace *f= ...;
Vcg::Point3d Dir1=f->PD1();
Vcg::Point3d Dir2=f->PD2();
```

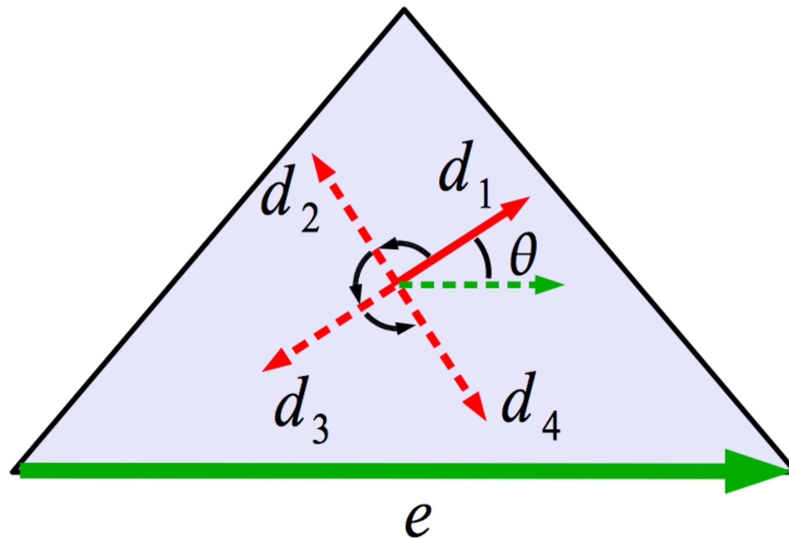
Draw

```
#include <wrap/gl/gl_field.h>
vcg::GLField<MyTriMesh>::GLDrawFaceField(m);
vcg::GLField<MyTriMesh>::GLDrawSingularity(m);
```



Cross Field(VCG)

- Invariance to 90° Rotations
  - **One** among d1,d2,d3,d4 identify the cross field
- Can be expressed using a single scalar θ



- Operations must take in consideration of 90° Invariance

Cross Field(VCG)

Other useful fuctions on cross field:

vcg/complex/algorithms/parametrization/tangent_field_operators.h

singularity

```
static void UpdateSingularByCross(MeshType &mesh)

typename MeshType::template PerVertexAttributeHandle<bool> Handle_Singular;
typename MeshType::template PerVertexAttributeHandle<int> Handle_SingularIndex;

Handle_Singular=vcg::tri::Allocator<MeshType>::template
GetPerVertexAttribute<bool>(mesh,std::string("Singular"));

Handle_SingularIndex =vcg::tri::Allocator<MeshType>::template
GetPerVertexAttribute<int>(mesh,std::string("SingularIndex"));

bool isSing=Handle_Singular[i];
int SingIndex=Handle_SingularIndex[i];
```

Interpolate

```
template < typename ScalarType >
vcg::Point3<ScalarType> InterpolateNRosy3D(. . .)\
```

Difference

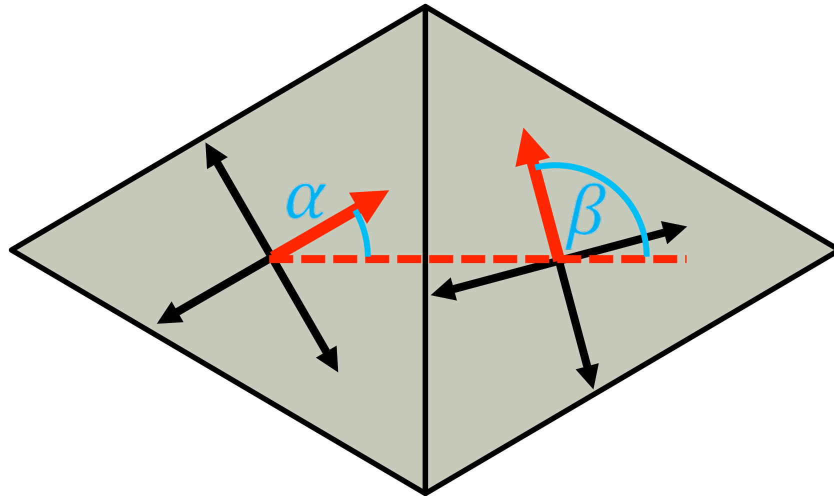
```
static typename FaceType::ScalarType DifferenceCrossField(. . . )
```

And many others…

Cross Field(Smooth)

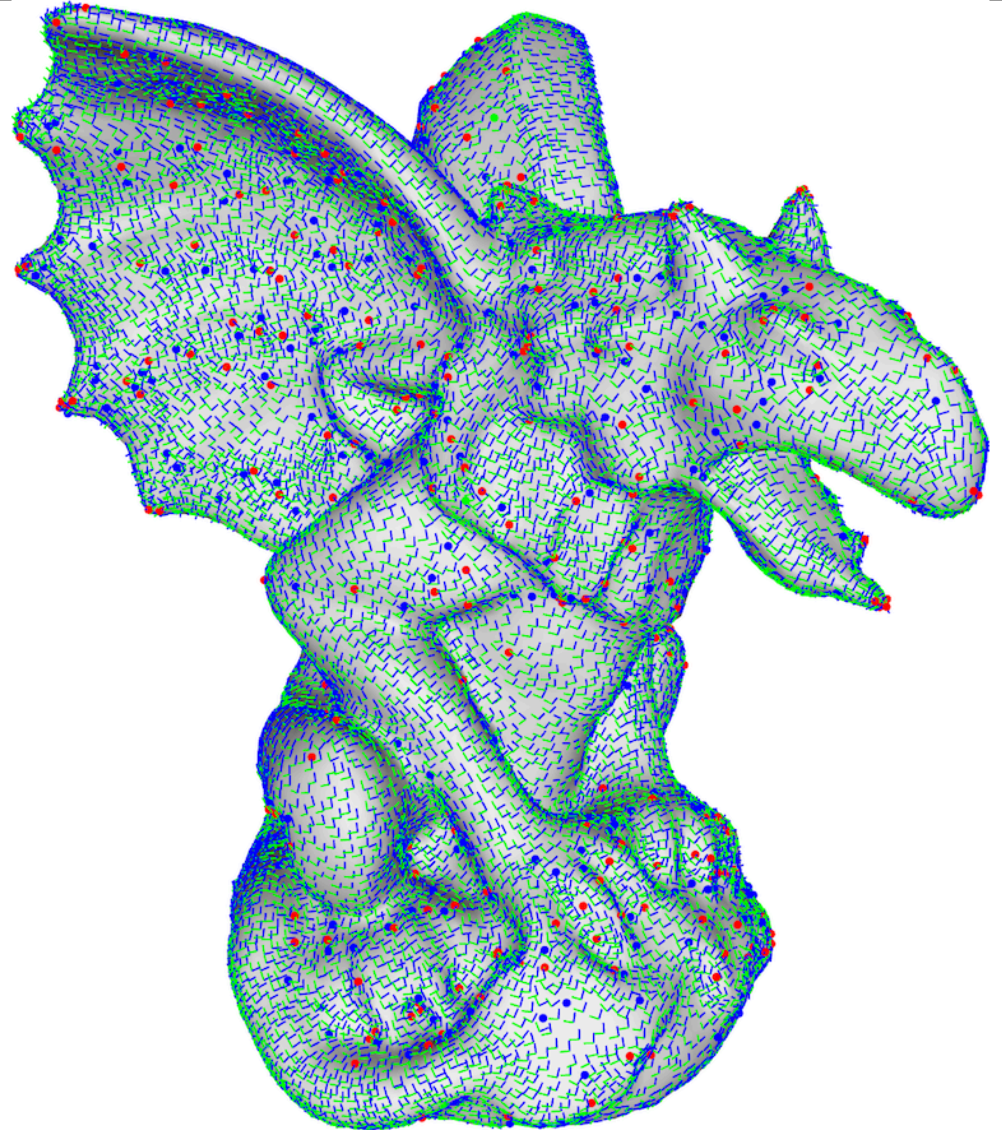
Smoothness measure: $E = \min_{k \in \mathbb{Z}} (\alpha - \beta + k \cdot 90°)^2$

Integer DOF for rotation



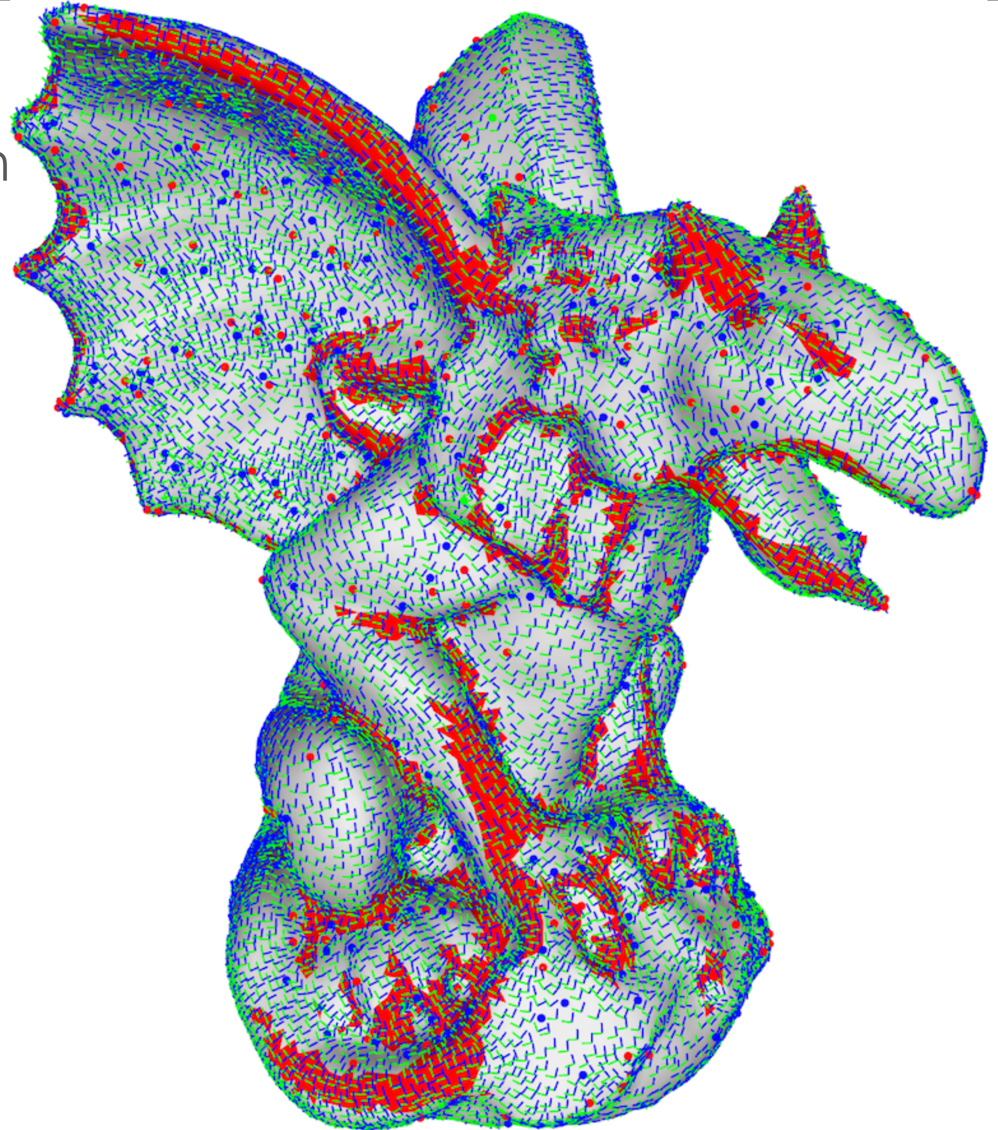
Cross Field(Smooth)

Compute curvature main directions at a given scale



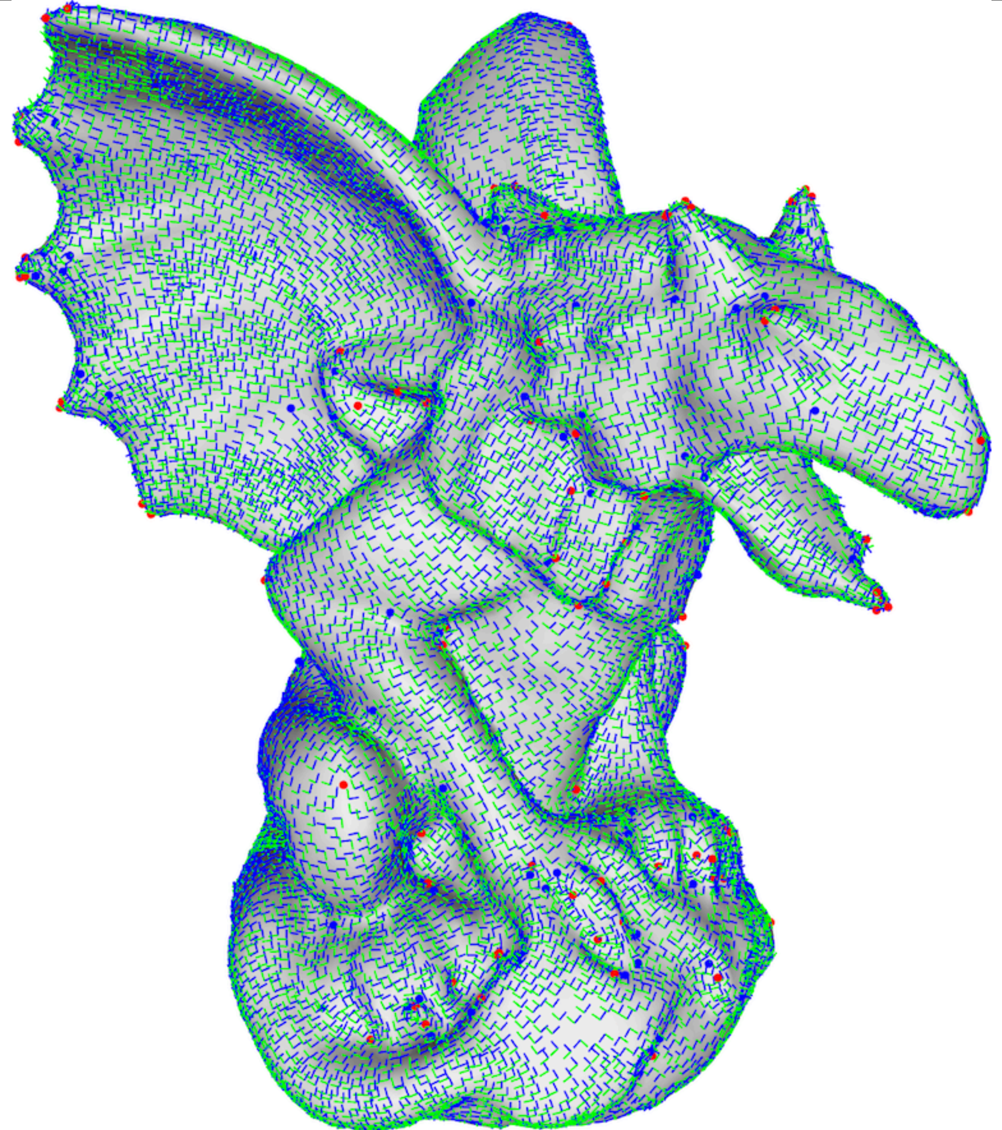
Cross Field(Smooth)

Select regions with with high curvature anisotropy



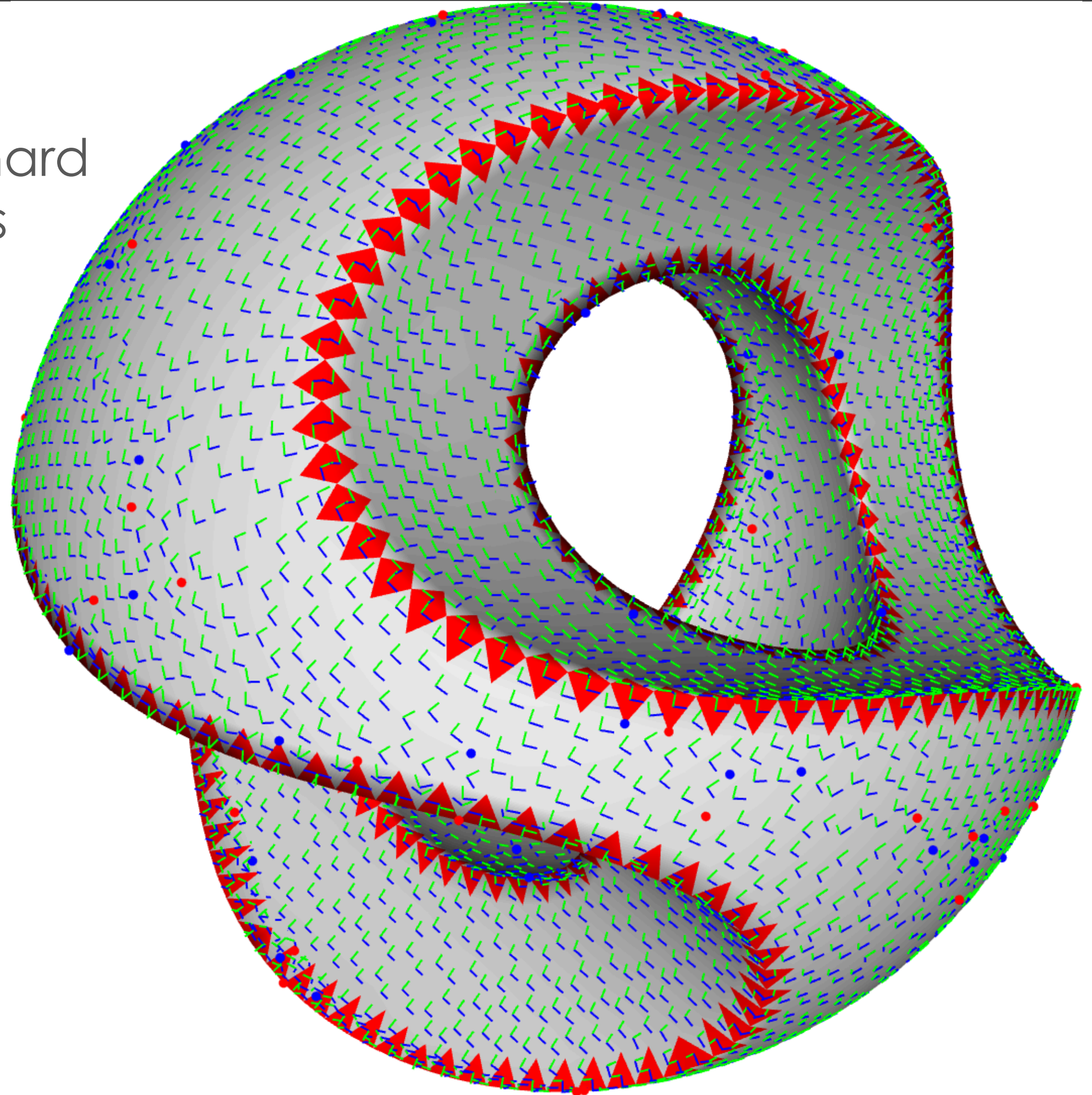
Cross Field(Smooth)

Use as hard constrants and smooth everywhere else



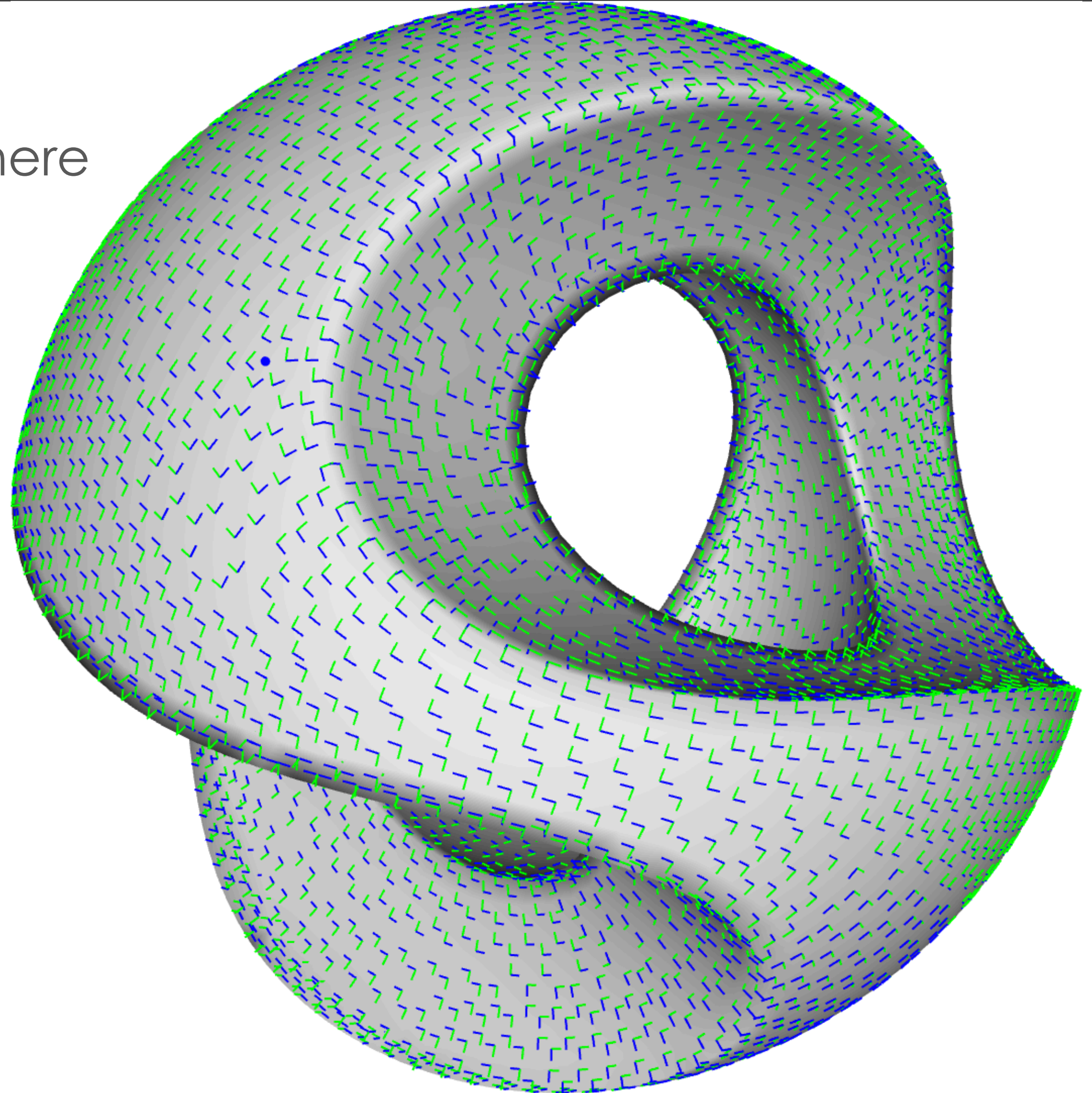
Cross Field(Smooth)

Use sharp features as hard constraints if availables



Cross Field(Smooth)

...and smooth everywhere else



Cross Field Computation(VCG)

Include

```
/Users/nicopietroni/Desktop/vcg/vcglib/wrap/igl/smooth_field.h
```

Parameter settings

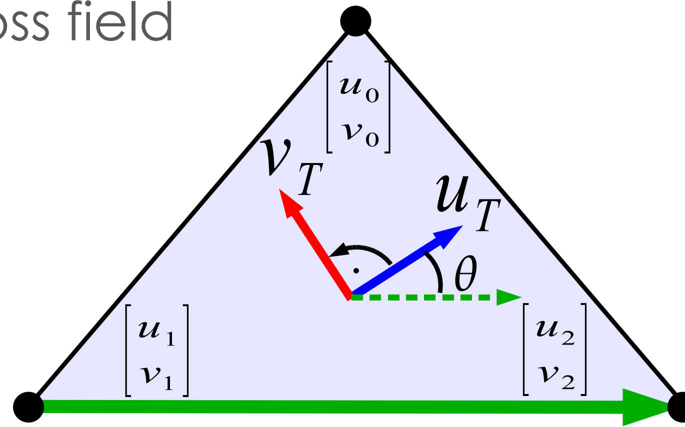
```
struct SmoothParam {
//the 90° rotation independence while smoothing the direction field
int Ndir;
//the weight of curvature if doing the smoothing keeping the field close to the original one
ScalarType alpha_curv;
//align the field to border or not
bool align_borders;
//threshold to consider some edge as sharp feature and to use as hard constraint (0, not use)
ScalarType sharp_thr;
//threshold to consider some edge as high curvature anisotropyand to use as hard constraint
ScalarType curv_thr;
//the method used to smooth MIQ or "Designing N-PolyVector Fields with Complex Polynomials"
SmoothMethod SmoothM;
//the number of faces of the ring used ot esteem the curvature
int curvRing;
};
```

call

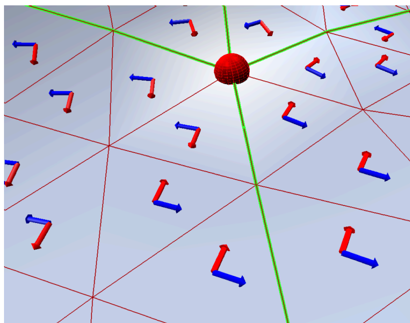
```
MyTriMesh tri_mesh
typedef FieldSmoother<MyTriMesh> FieldSmootherType;
FieldSmootherType::SmoothParam SParam;
FieldSmootherType::SmoothDirections(tri_mesh,SParam);

vcg::tri::CrossField<MyTriMesh>::OrientDirectionFaceCoherently(tri_mesh);
vcg::tri::CrossField<MyTriMesh>::UpdateSingularByCross(tri_mesh);
```

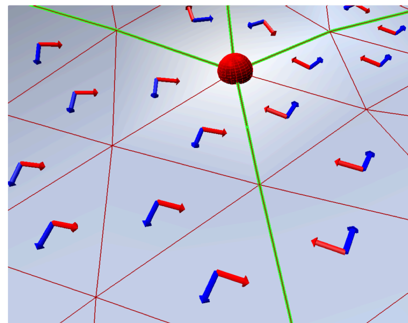
Field Aligned Parametrization

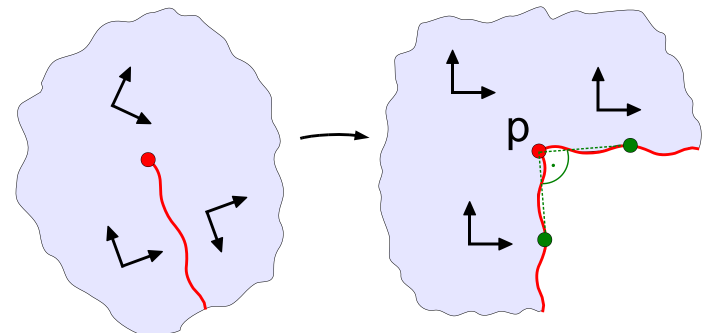
Align triangles to cross field



In the computation two linear scalar functions (u, v) are sought whose gradients are ori- ented consistently with the cross field directions.

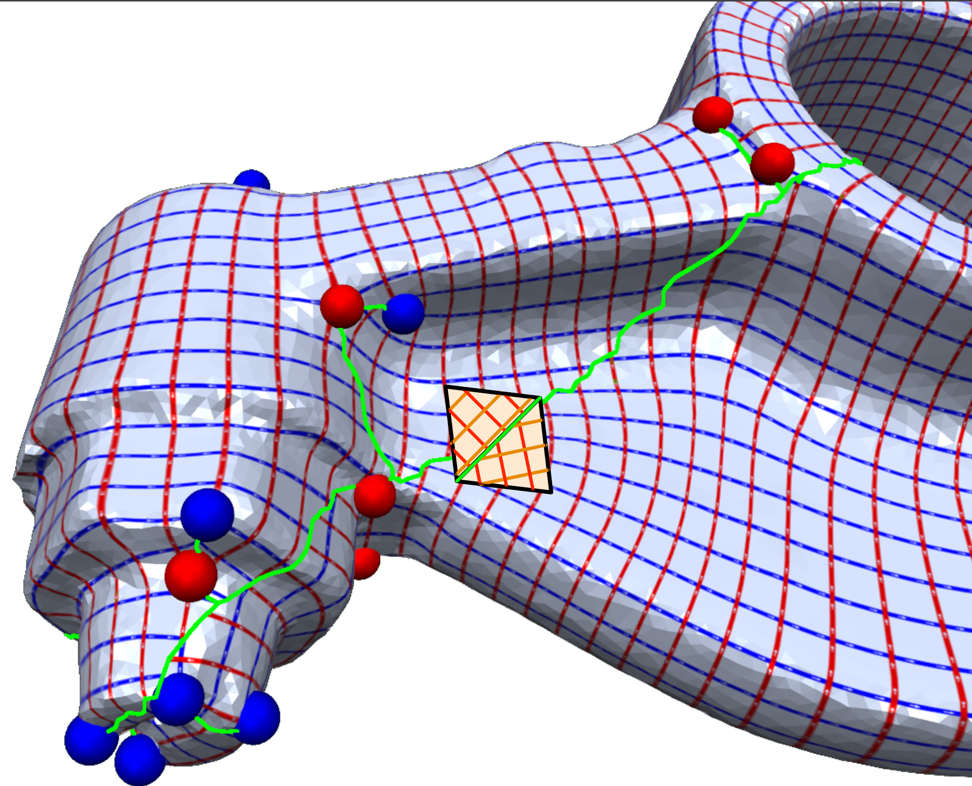


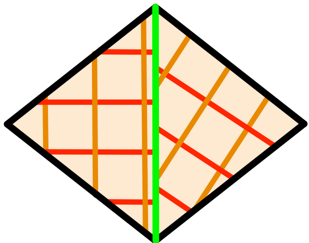




Handling cuts

- Discontinuous integer-lines at cut edges

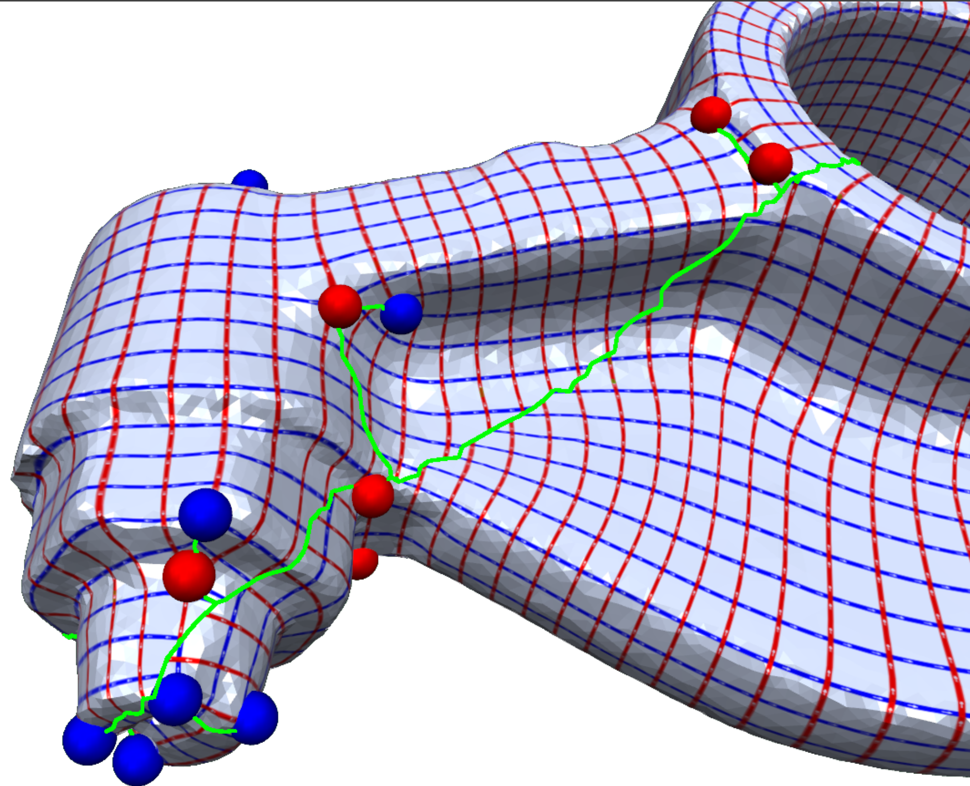


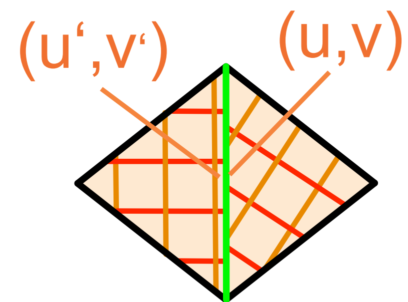


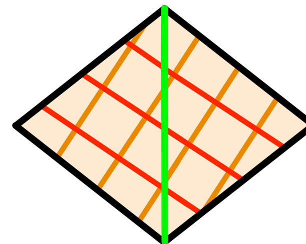
Handling cuts

- Discontinuous integer-lines at cut edges

- Continuity constraint

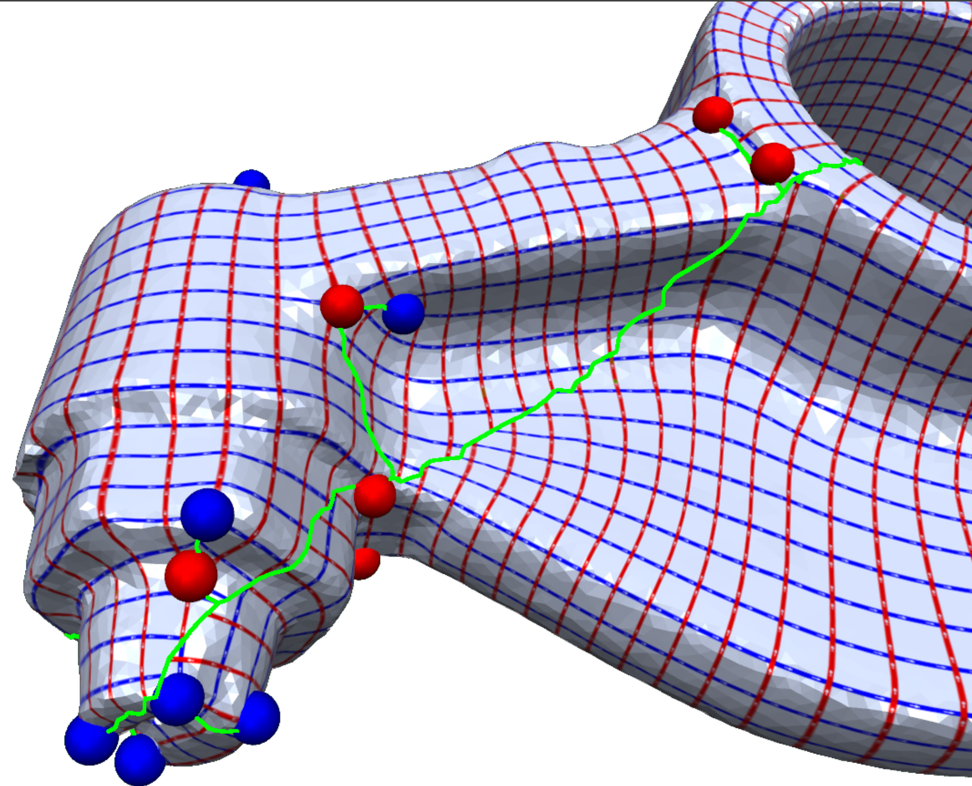


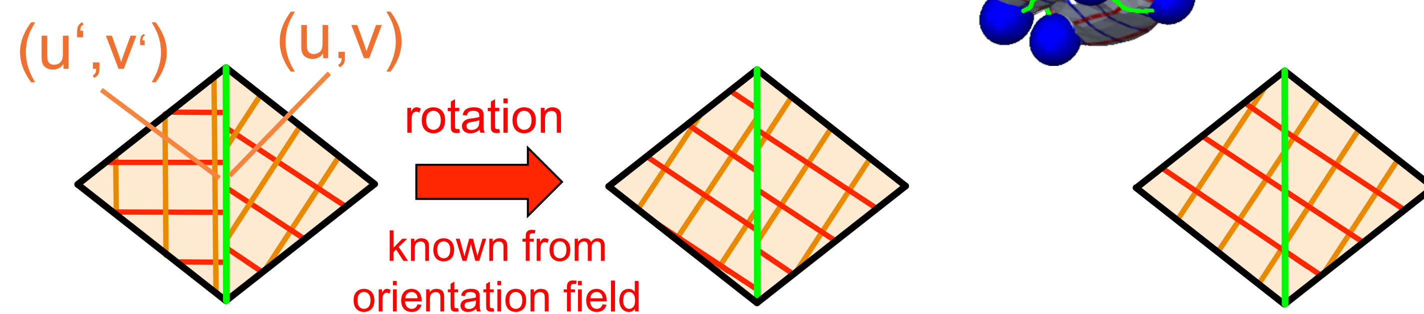




Handling cuts

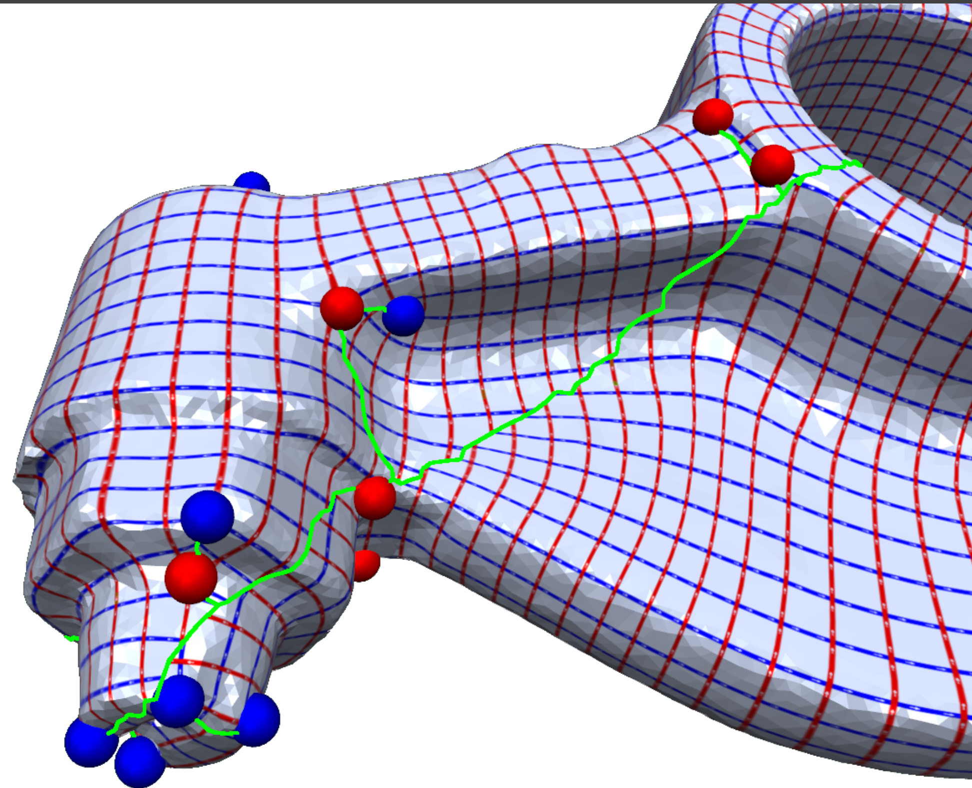
- Discontinuous integer-lines at cut edges
- Continuity constraint

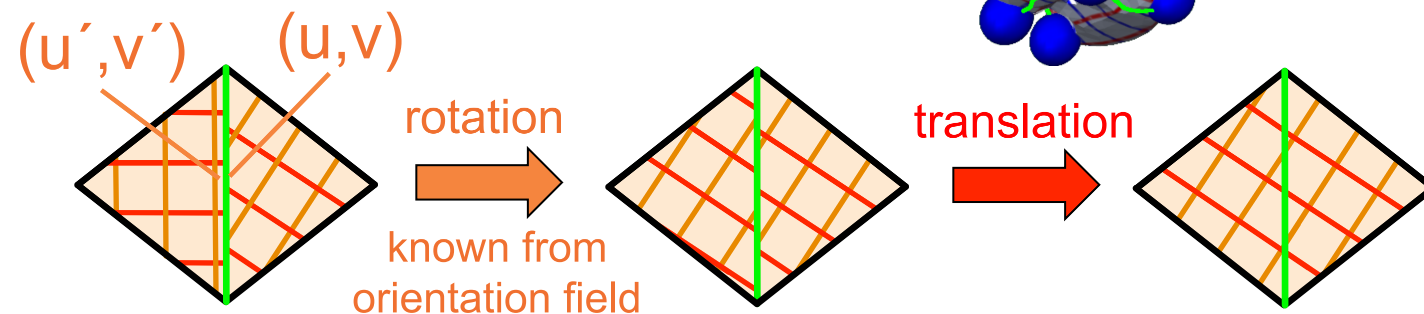




Handling cuts

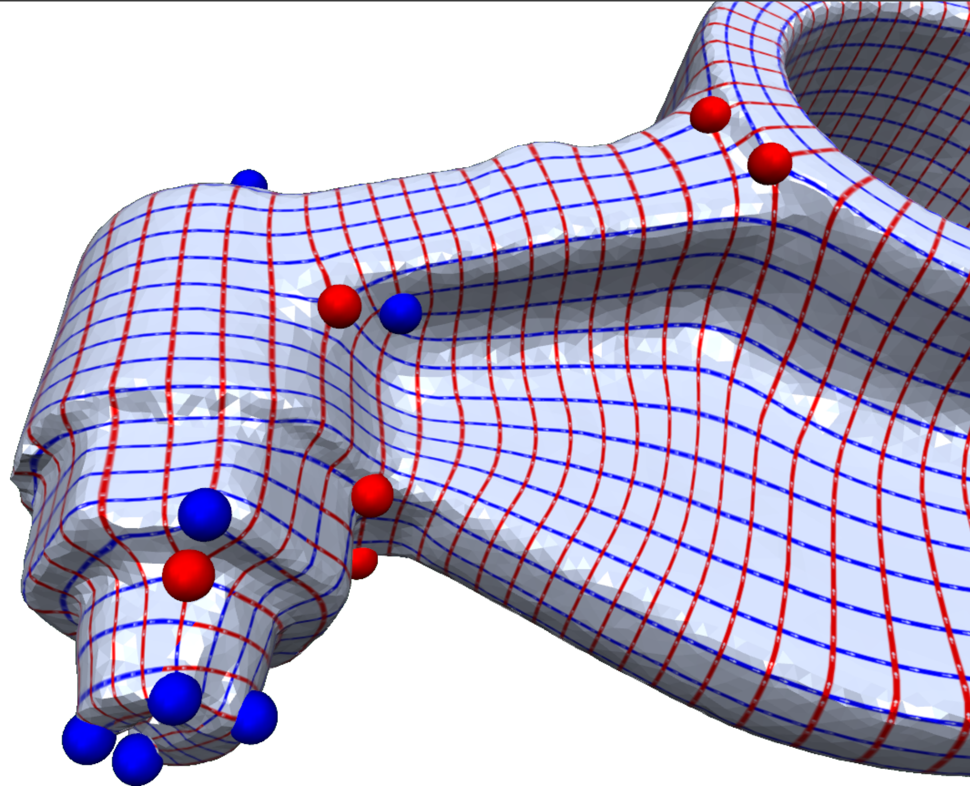
- Discontinuous integer-lines at cut edges
- Continuity constraint

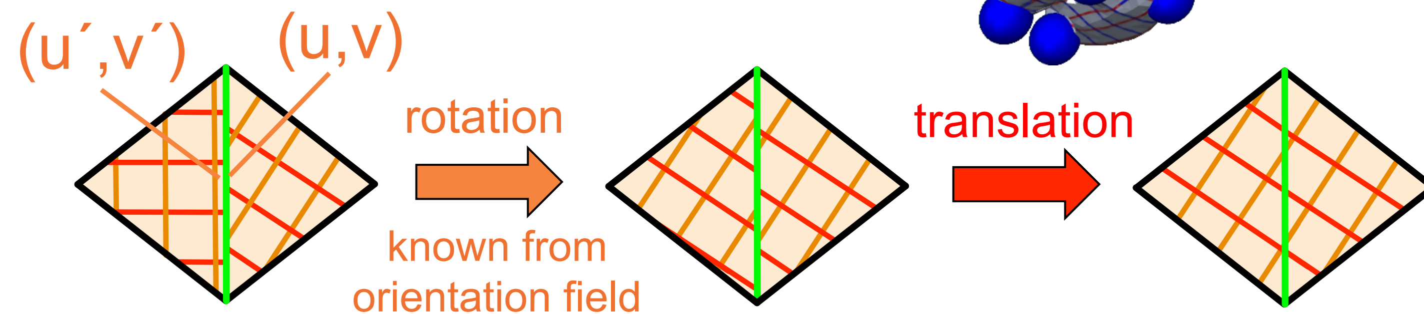




Handling cuts

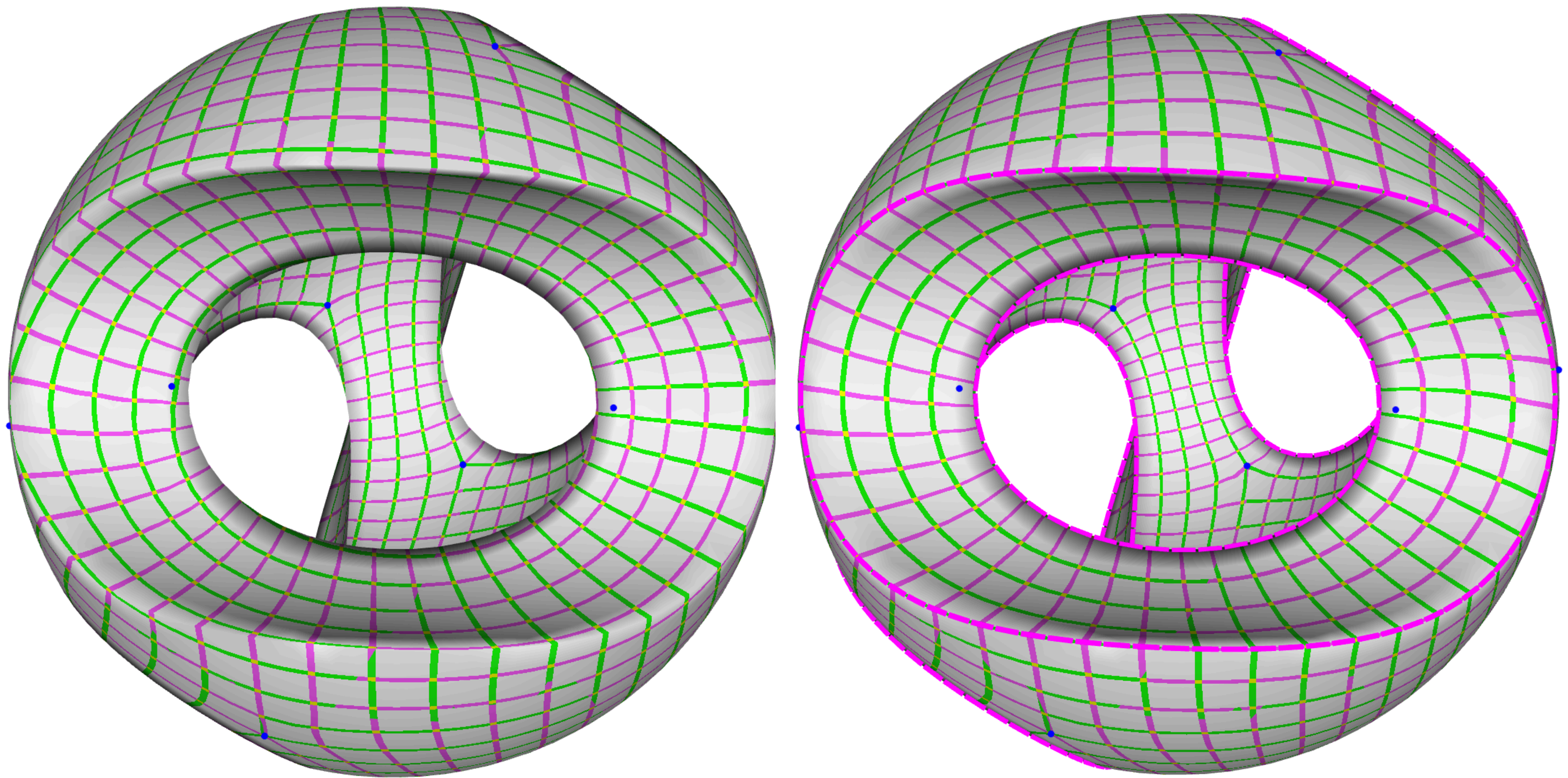
- Discontinuous integer-lines at cut edges
- Continuity constraint





Sharp Features

Sharp features as hard constraints



Mixed Integer Parametrization(VCG)

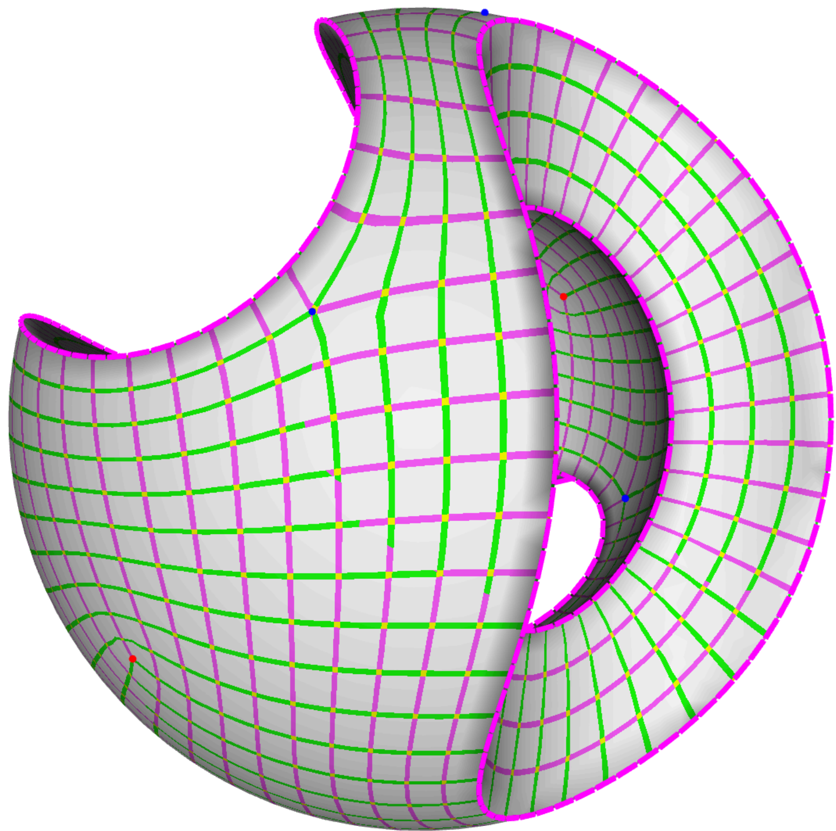
Parametrizator in wrap/igl/miq_parametrization.h

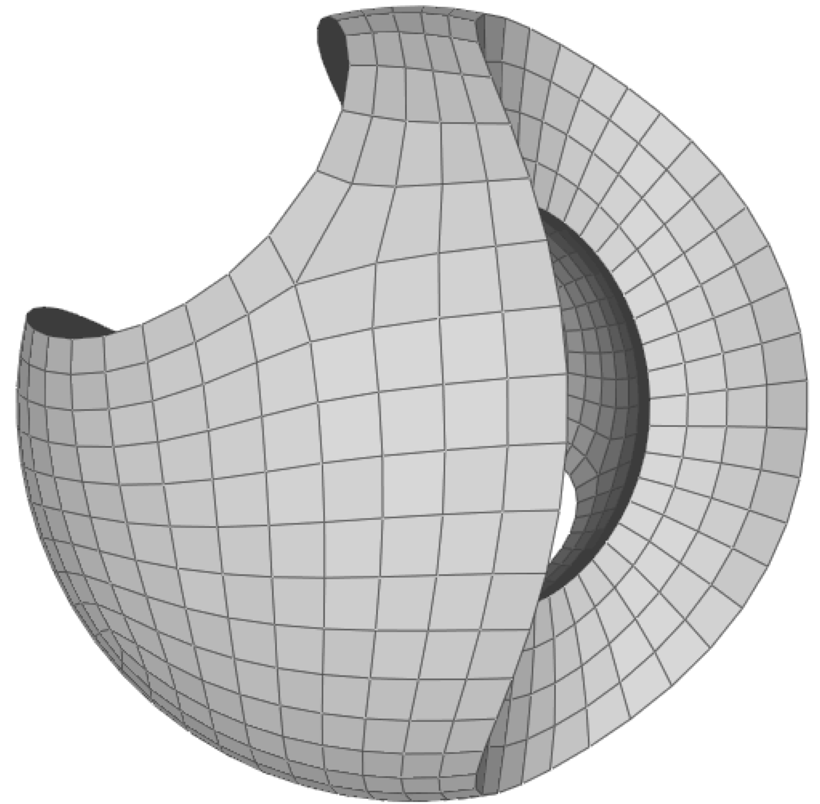
```
struct MIQParameters
    {
        //the gradient of the parametrization 1 is the bb diagonal..
        double gradient;
        //do the rounding or not across cuts... set to true to get a quadrangulation
        bool doRound;
       ...
        double crease_thr;
        //number of 90° rotation independence
        int Ndir;
        //round or not the singularities
        bool round_singularities;
        //use the crease edges as feature or not
        bool crease_as_feature;
};
```

A simple call

```
MyTriMesh tri_mesh;
vcg::tri::MiQParametrizer<MyTriMesh>::MIQParameters MiqP;
vcg::tri::MiQParametrizer<MyTriMesh>::MIQParametrize(tri_mesh,MiqP);
```

Quadrangulation





Quadrangulation(VCG)

- Extension of the refinement algorithm
- in <vcg/complex/algorithms/quadrangulator.h>
- Cut trought integer lines
- Split each triangle
- Merge edges/triangles to quads

```
MyTriMesh tri_mesh;
MyQuadMesh quad_mesh;
std::vector< std::vector< short int> > AxisUV
vcg::tri::Quadrangulator<MyTriMesh,MyQuadMesh> Quadr;
Quadr.Quadrangulate(tri_mesh,quad_mesh,AxisUV);
```

FGT: 2015/2016
Parametrization & Remeshing

Nico Pietroni (nico.pietroni@isti.cnr.it)

ISTI – CNR