libigl Prototyping Geometry Processing Research in C++

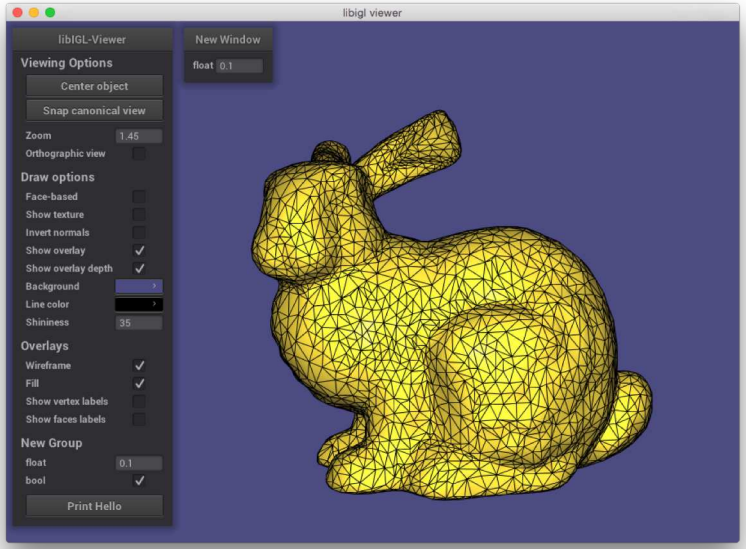
Alec Jacobson
University of Toronto

https://github.com/libigl/libigl-course

Daniele Panozzo
New York University

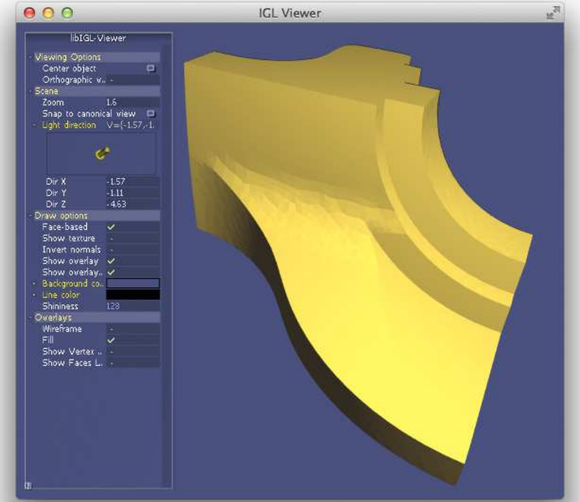
libigl is amassing a large library of C++ implementations of core routines

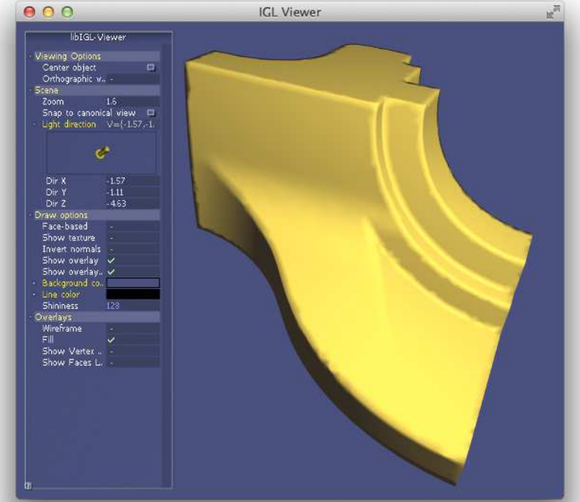
- mesh I/O (.obj, .stl, .off, .ply, .wrl, .mesh)

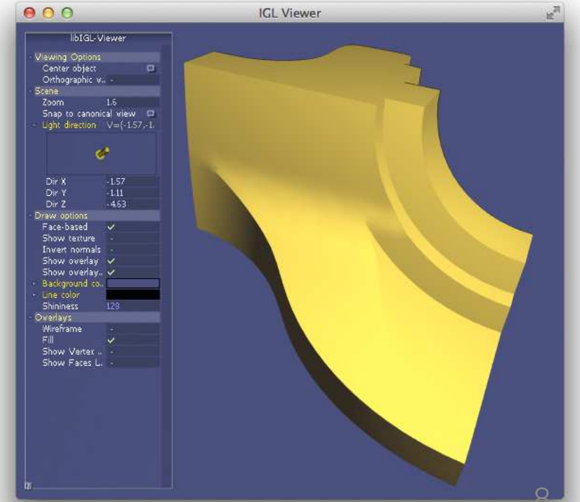


libigl is amassing a large library of C++ implementations of core routines

- mesh I/O (.obj, .stl, .off, .ply, .wrl, .mesh)
- areas, normals, angles, edge lengths, adjacencies





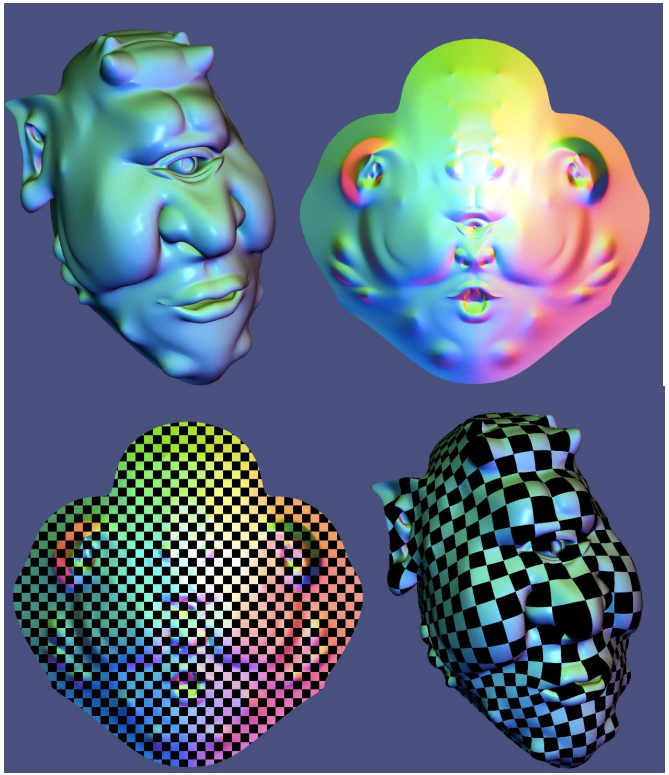


libigl is amassing a large library of C++ implementations of core routines

- mesh I/O (.obj, .stl, .off, .ply, .wrl, .mesh)
- areas, normals, angles, edge lengths, adjacencies
- cotangent Laplacian, mass matrix, gradient, divergence

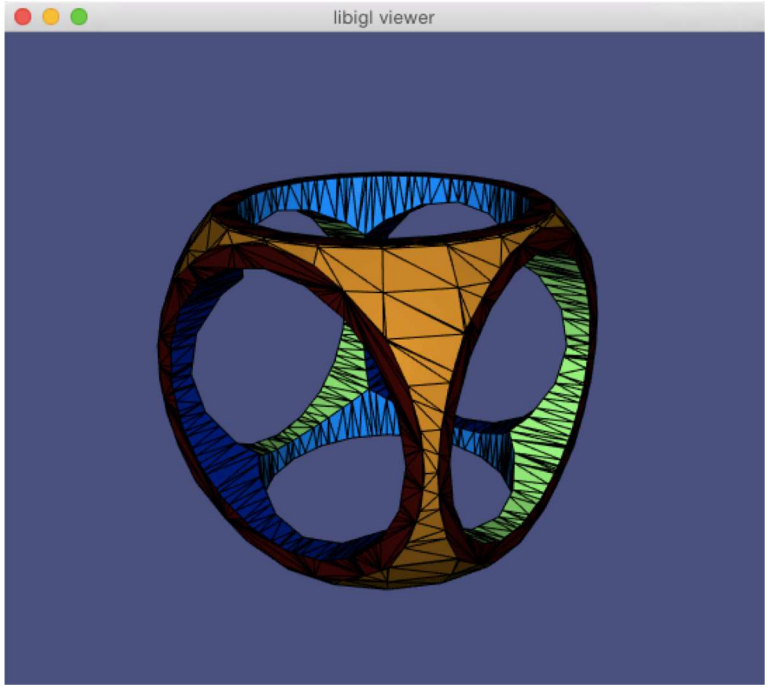
libigl is amassing a large library of C++ implementations of core routines

- mesh I/O (.obj, .stl, .off, .ply, .wrl, .mesh)
- areas, normals, angles, edge lengths, adjacencies
- cotangent Laplacian, mass matrix, gradient, divergence
- parameterization, smoothing, deformation, decimation



libigl is amassing a large library of C++ implementations of core routines

- mesh I/O (.obj, .stl, .off, .ply, .wrl, .mesh)
- areas, normals, angles, edge lengths, adjacencies
- cotangent Laplacian, mass matrix, gradient, divergence
- parameterization, smoothing, deformation, decimation
- boolean operations, quad meshing, closest point queries



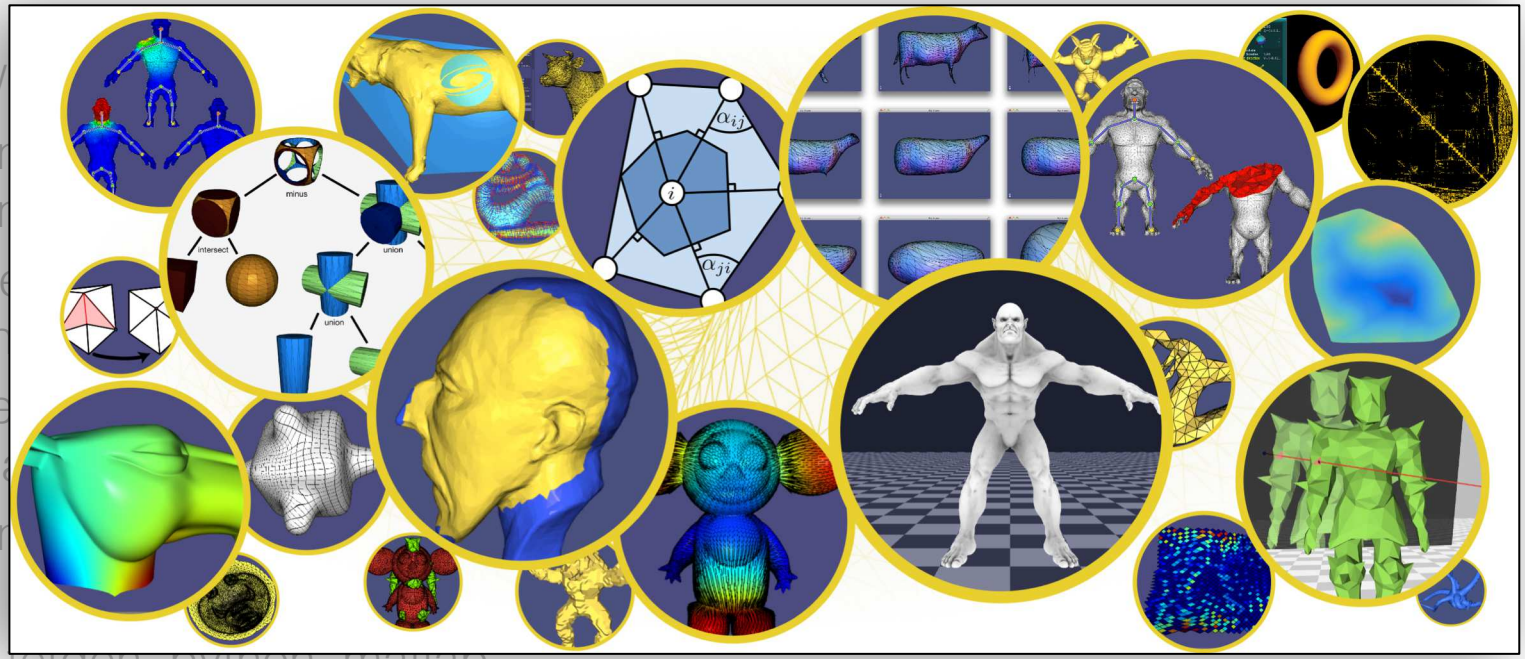
libigl is amassing a large library of C++ implementations of core routines

- mesh I/O (.obj, .stl, .off, .ply, .wrl, .mesh)
- areas, normals, angles, edge lengths, adjacencies
- cotangent Laplacian, mass matrix, gradient, divergence
- parameterization, smoothing, deformation, decimation
- boolean operations, quad meshing, closest point queries
- CGAL, tetgen, python, matlab

libigl

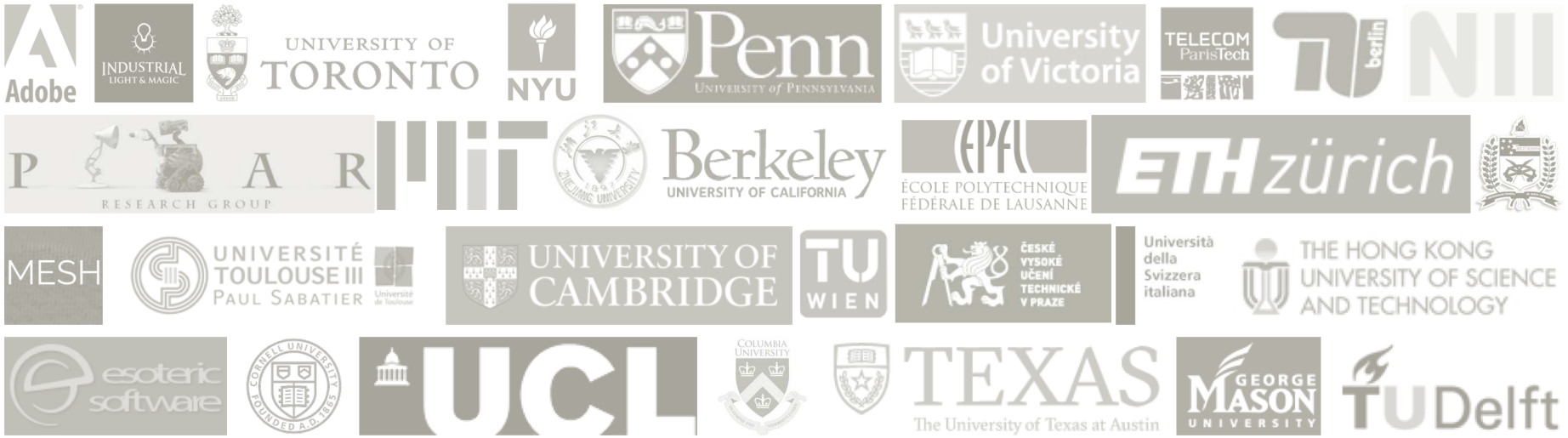
libigl is amassing a large library of C++ implementations of core routines

- mesh I/
- areas, r
  adjacen
- cotange
  gradien
- parame
  deforma
- boolean
  closest
- CGAL, tetgen, python, matlab
- *… much more*



libigl continues its mission to empower *research*

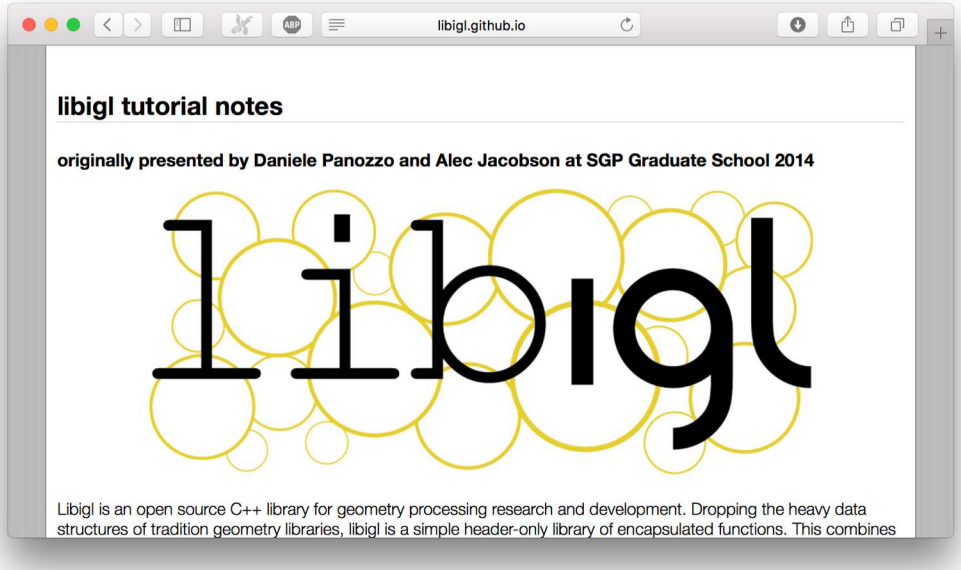
libigl continues its mission to empower *research*



Emphasis on fast prototyping and exploration, rather than app development

libigl hosts an online tutorial

libigl.github.io/libigl/tutorial/tutorial.html



Who are we?

Alec Jacobson

University of Toronto

Daniele Panozzo

New York University

James Zhou

Adobe Research

Winding Number
Mesh Booleans
Unit Testing

Jeremie Dumas

NYU/nTopology

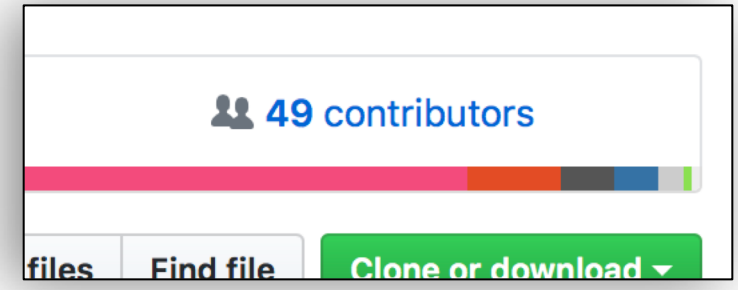
CMake Build System
Viewer

Sebastian Koch

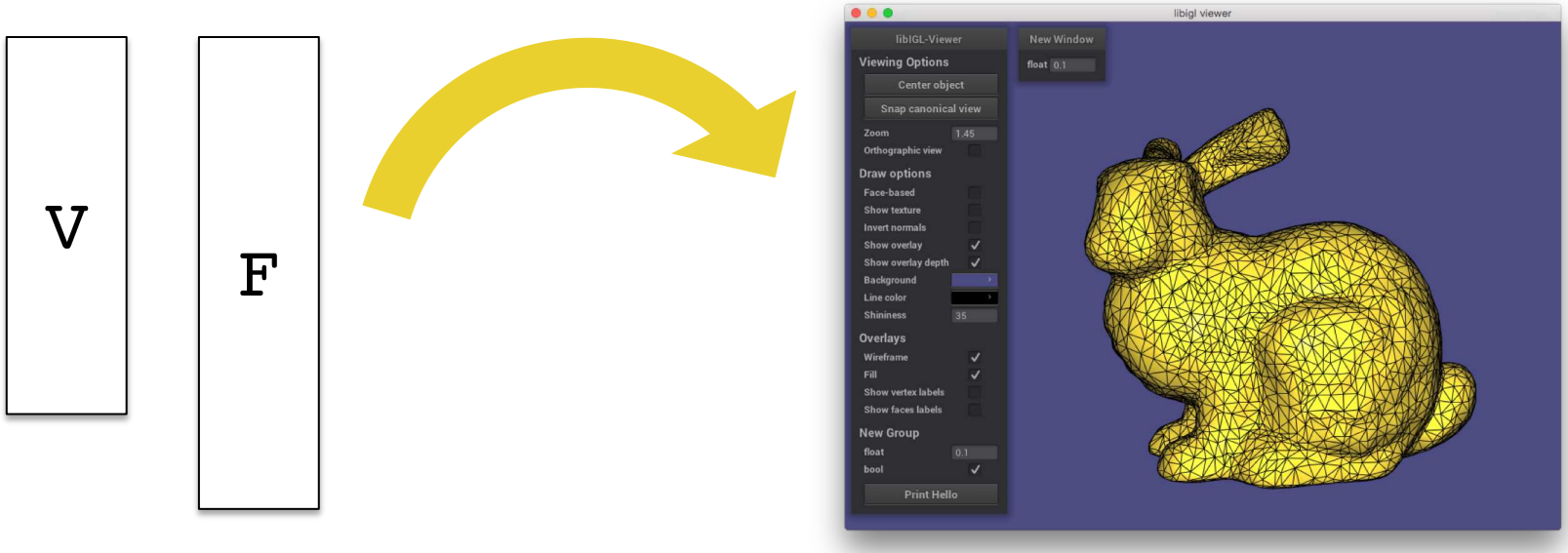
TU Berlin

Python Bindings
Javascript Port
(coming soon!)

Community Effort

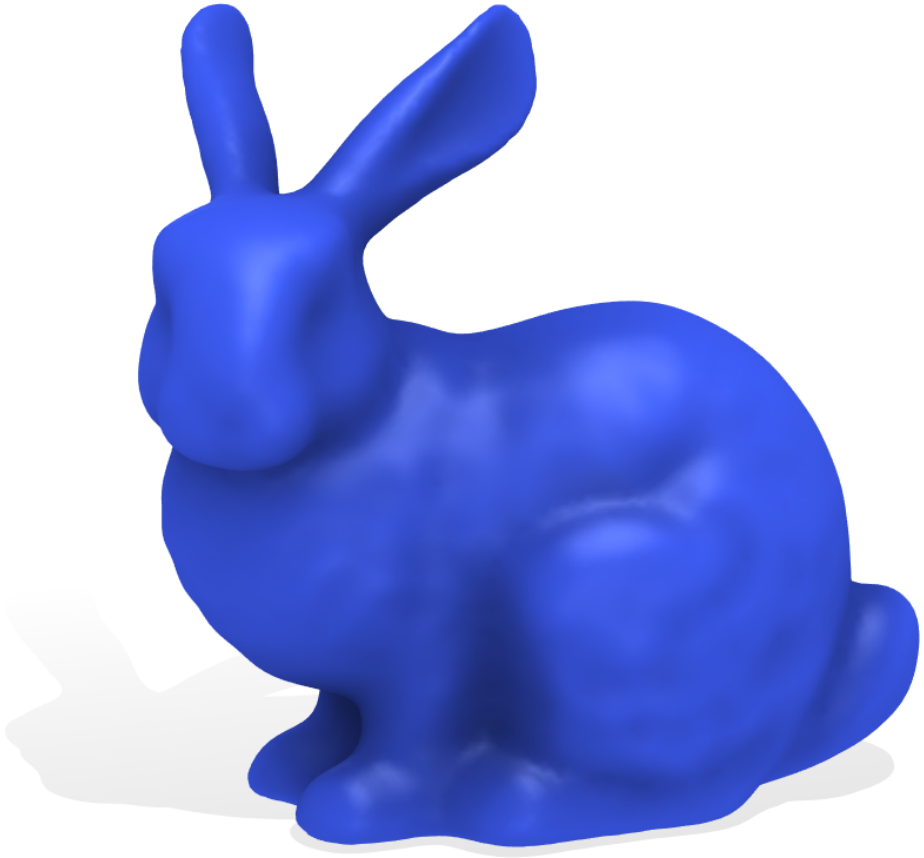


Christian Schüller,
Kevin Wallimann,
Stefan Brugger,
Yotam Gingold,
….

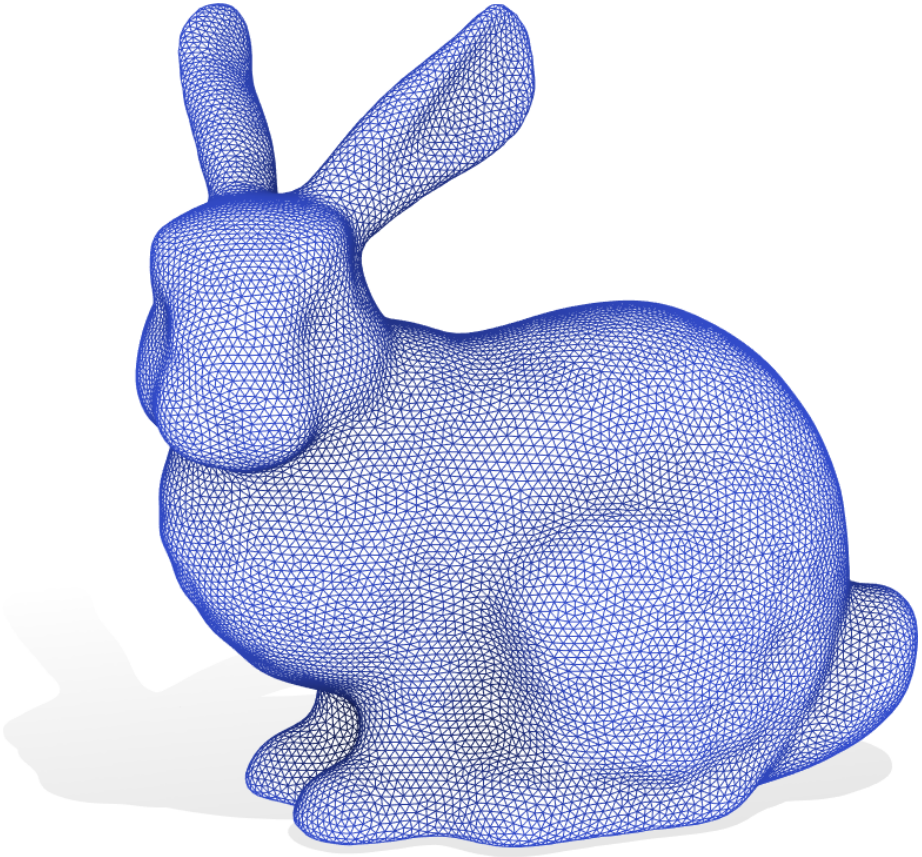


libigl data structures & style

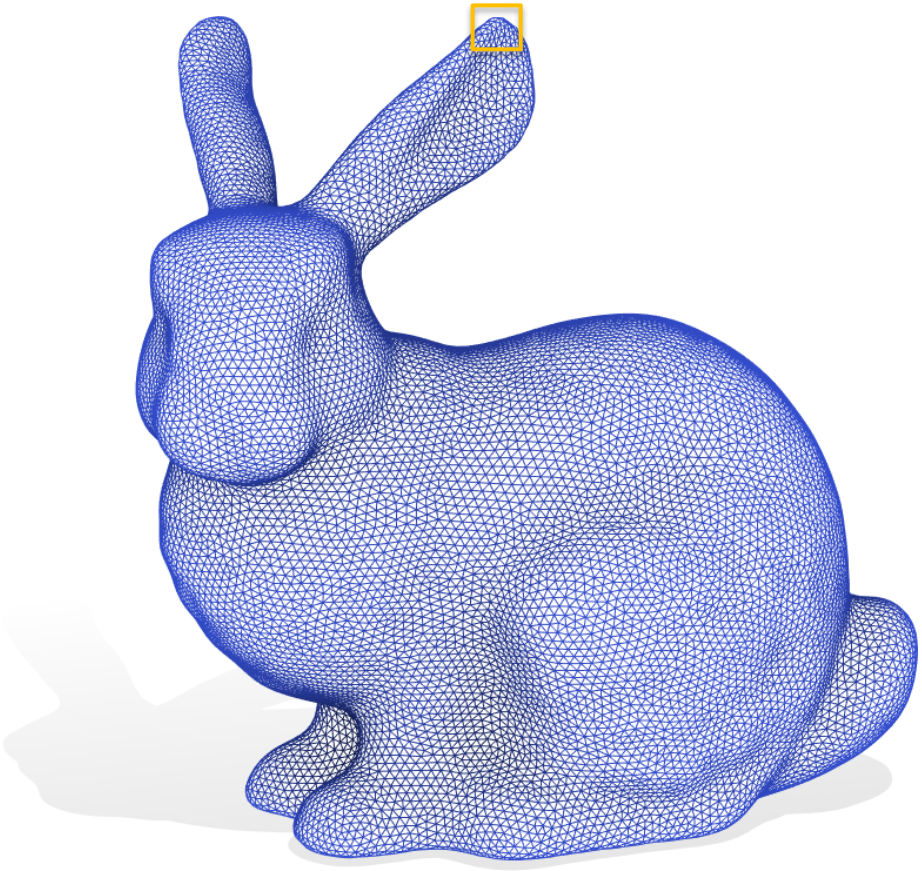
Triangle meshes discretize surfaces…

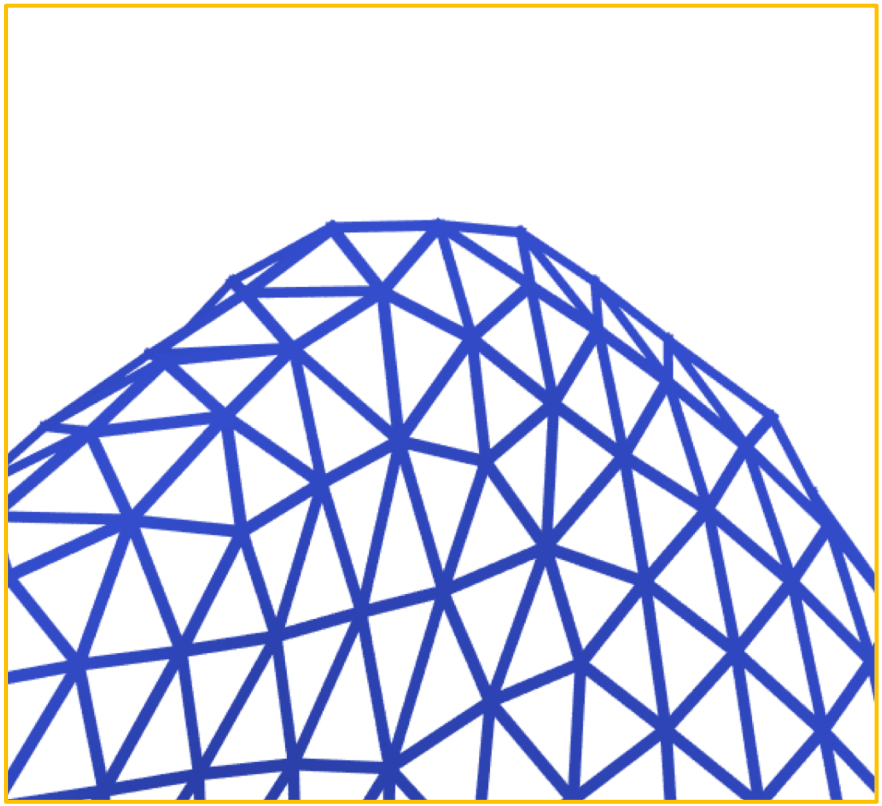


Triangle meshes discretize surfaces…

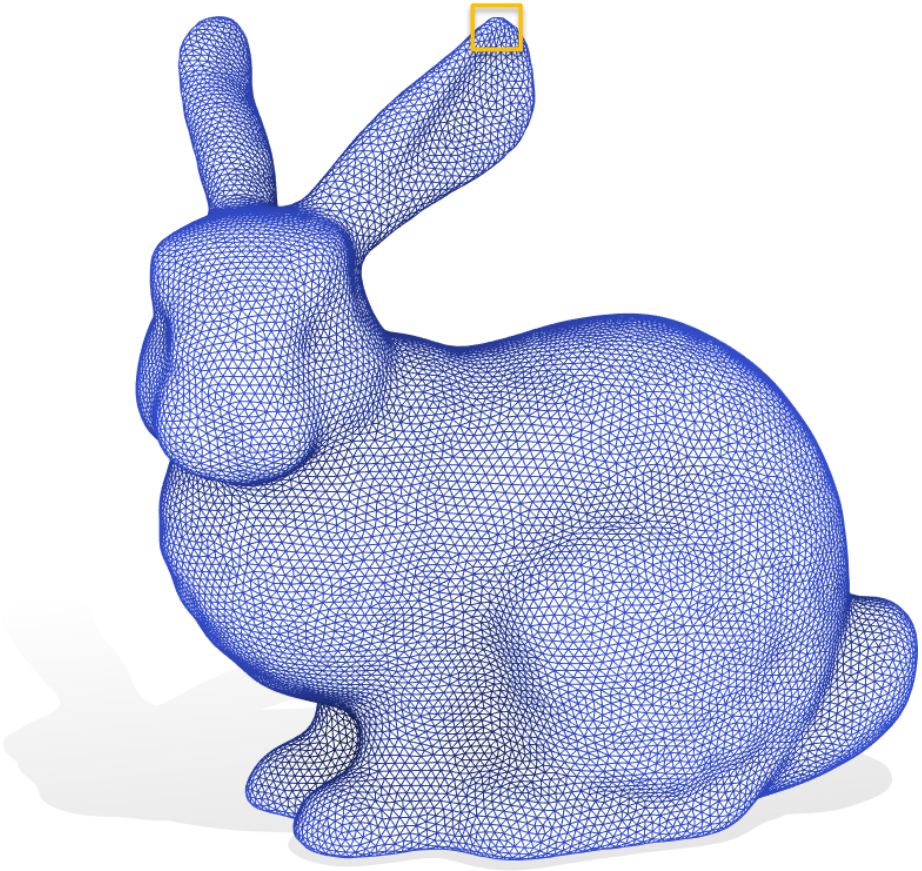


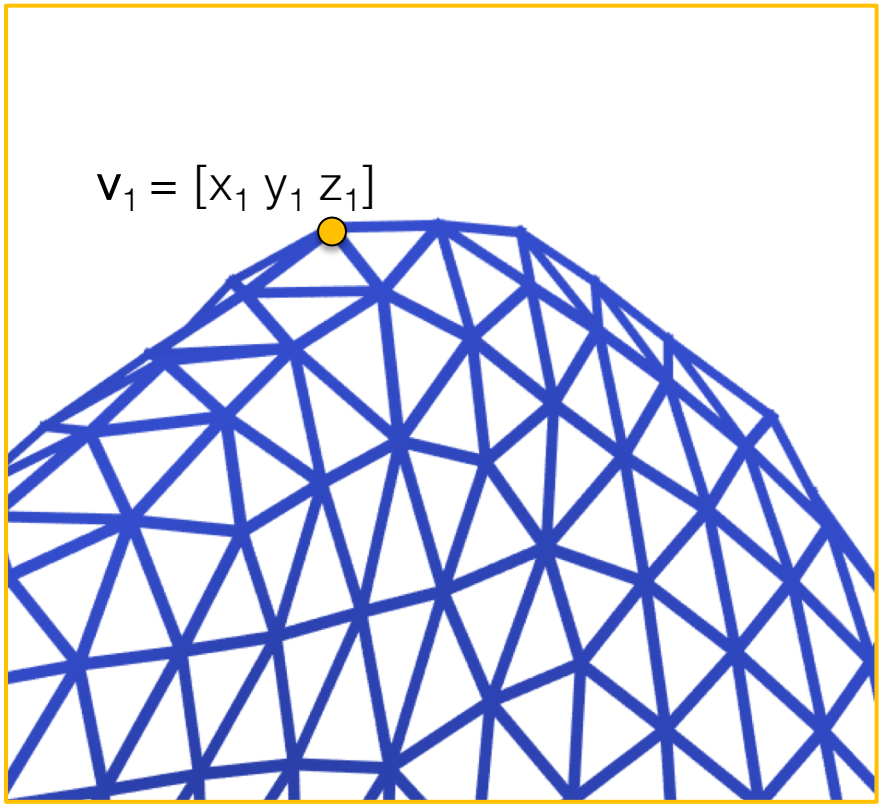
Triangle meshes discretize surfaces…



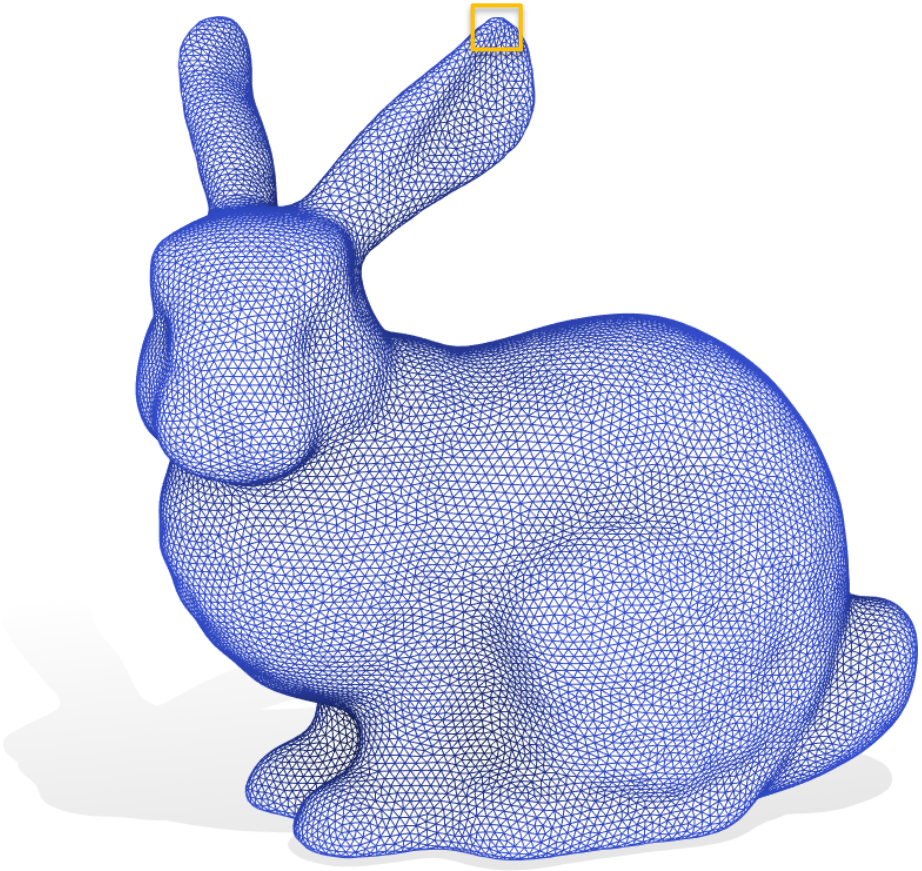


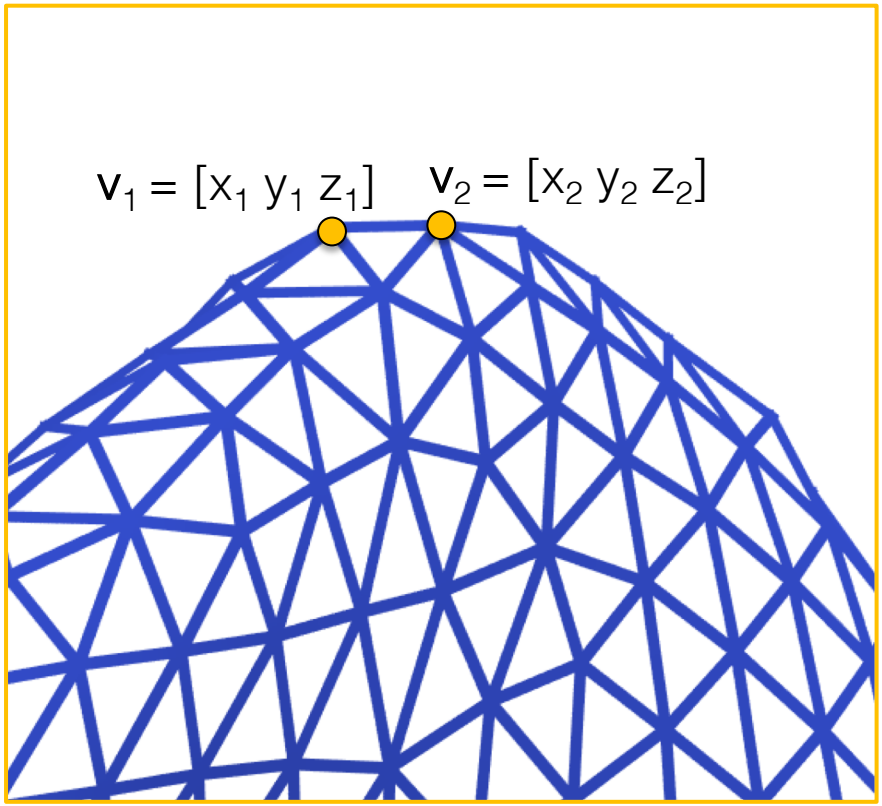
Triangle meshes discretize surfaces…



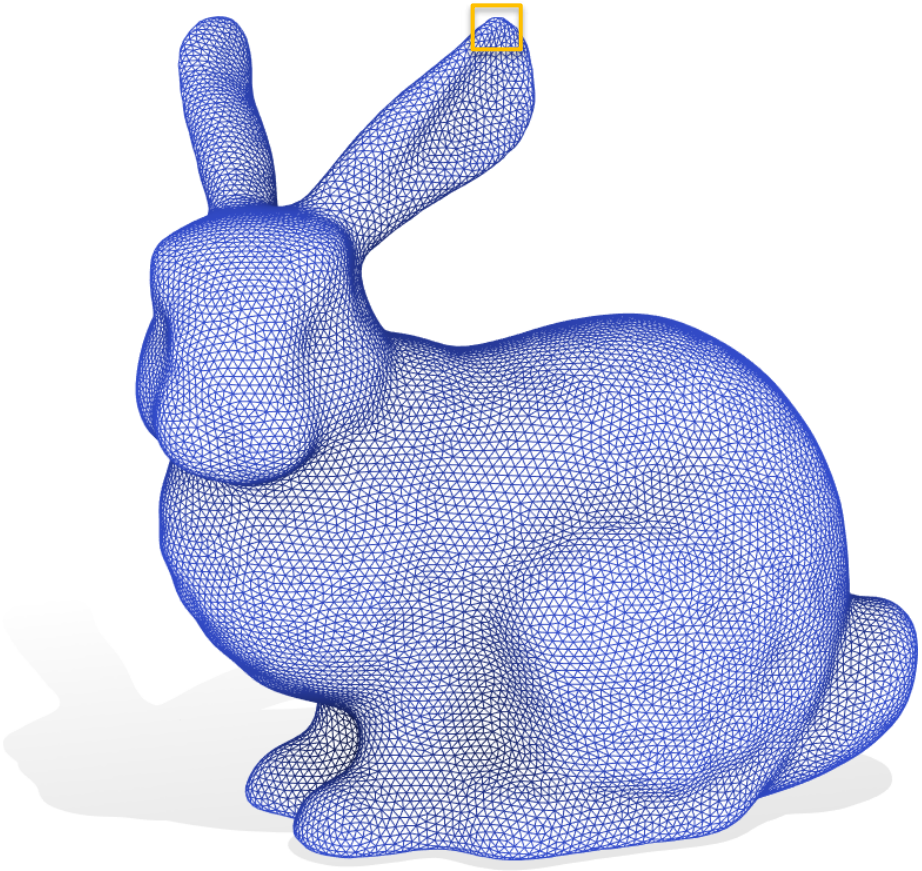


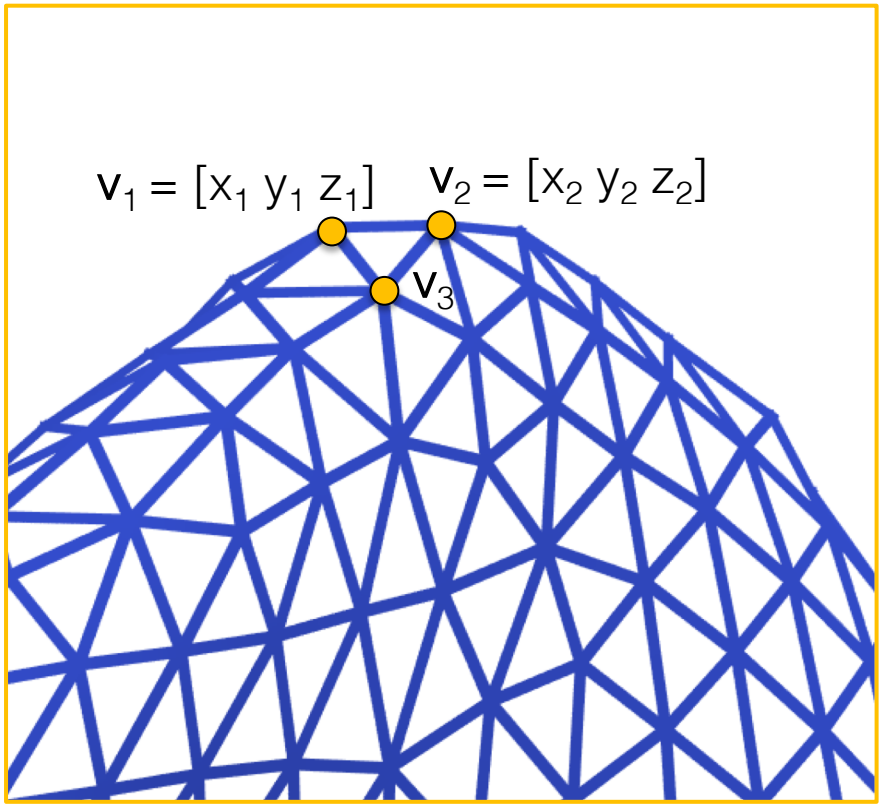
Triangle meshes discretize surfaces…



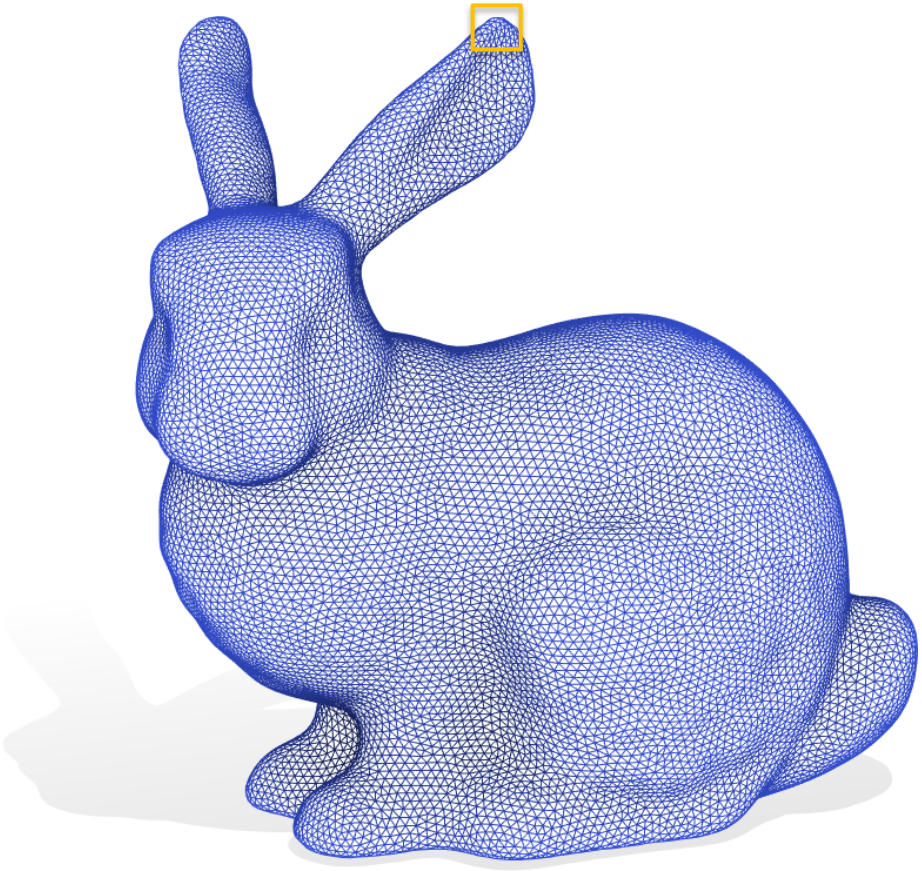


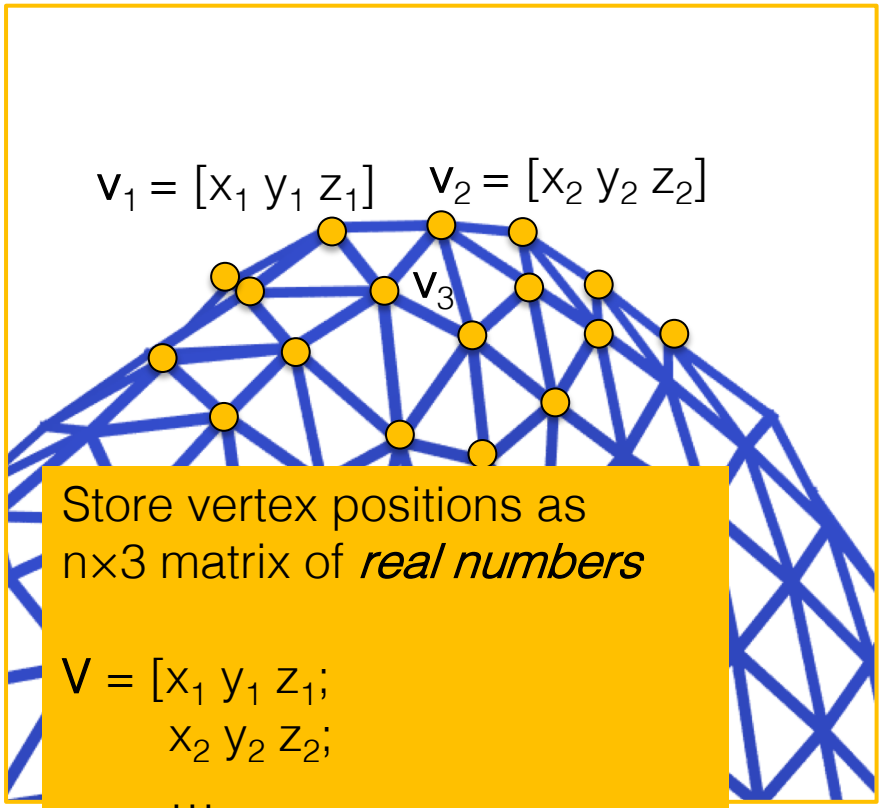
Triangle meshes discretize surfaces…





Triangle meshes discretize surfaces…

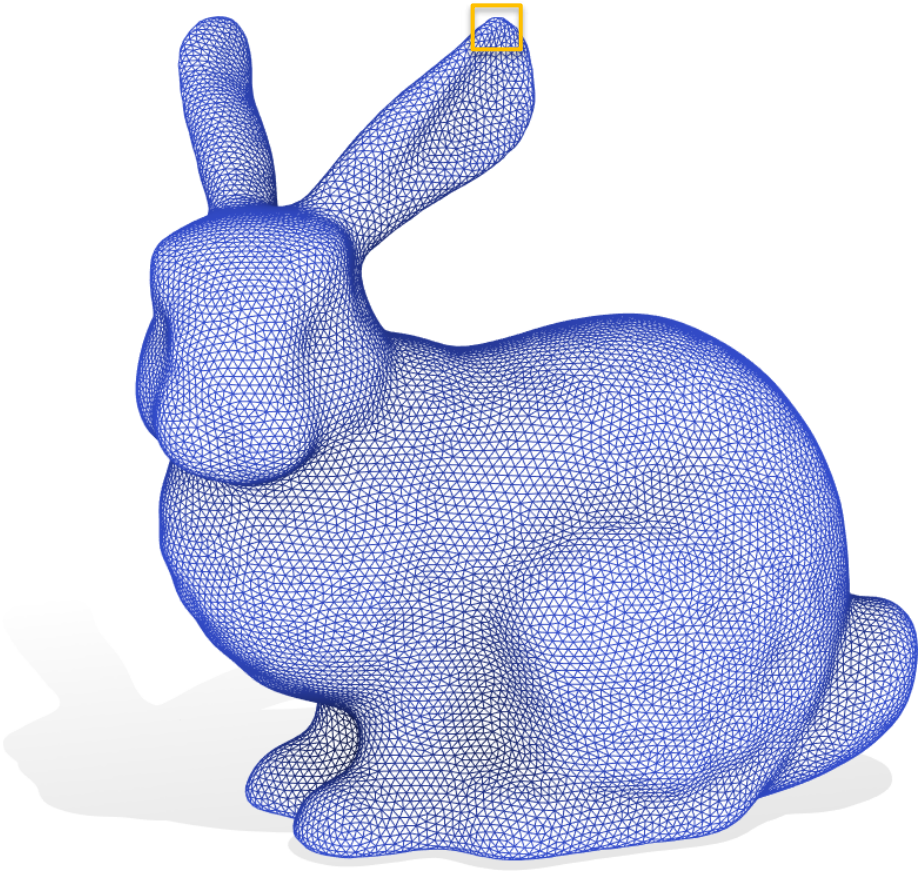


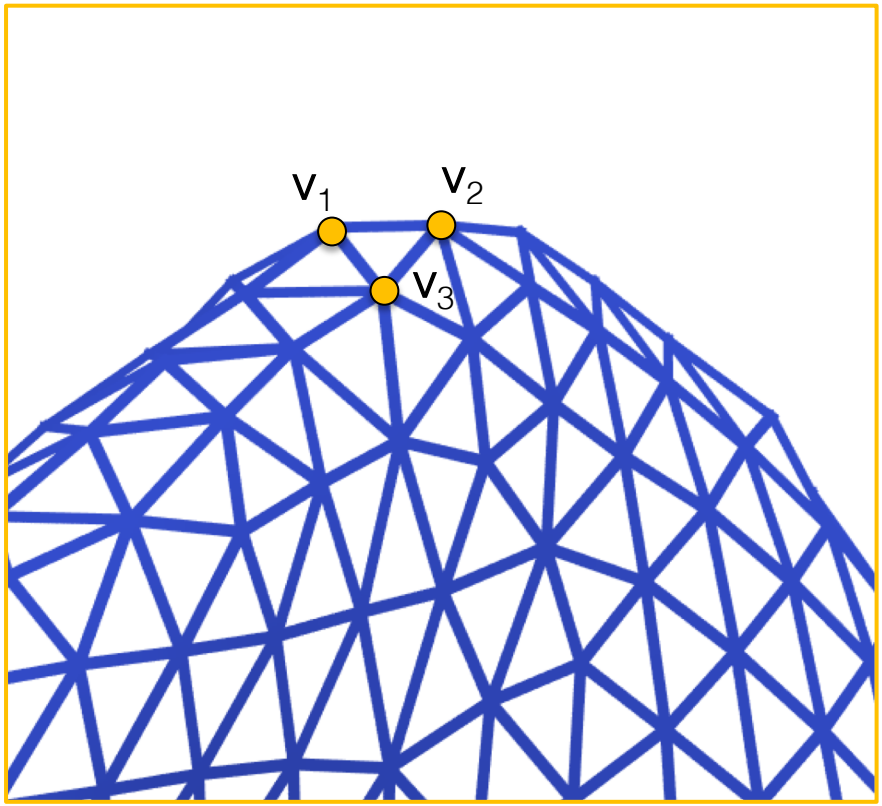


Store vertex positions as n×3 matrix of ***real numbers***

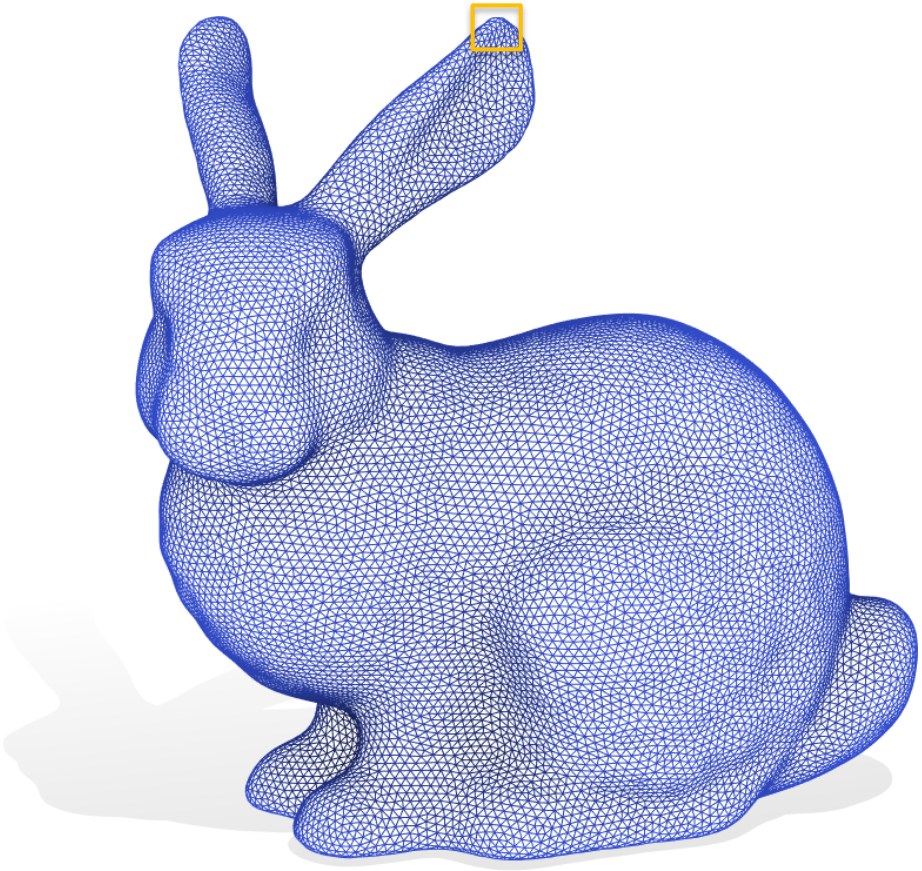
$$V = [x_1\ y_1\ z_1;\ x_2\ y_2\ z_2;\ \ldots\ x_n\ y_n\ z_n]$$

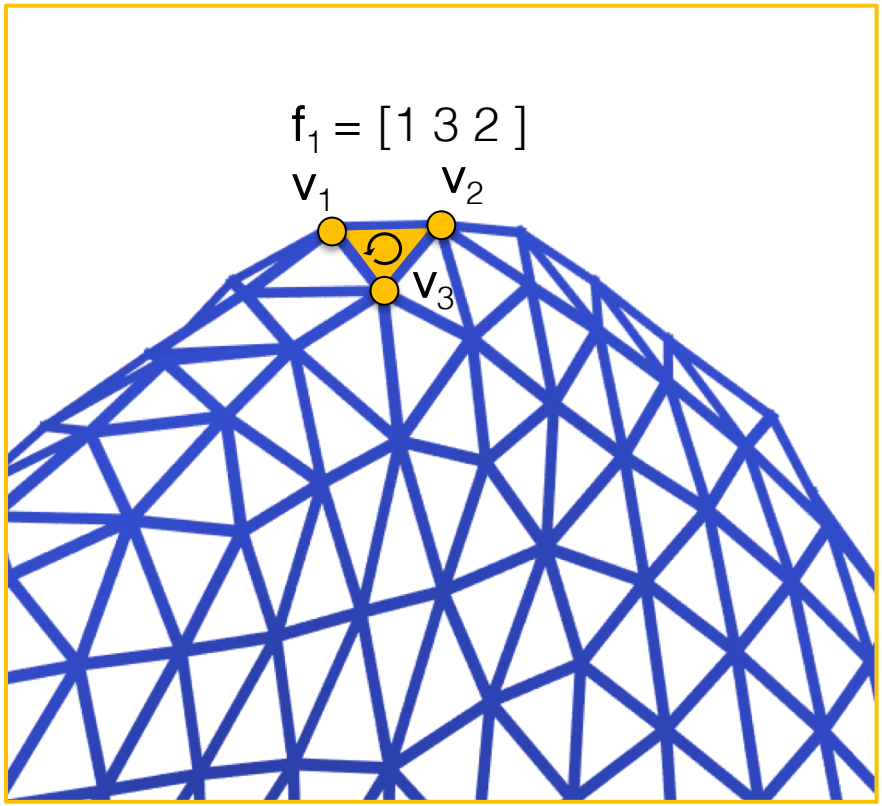
Triangle meshes discretize surfaces…



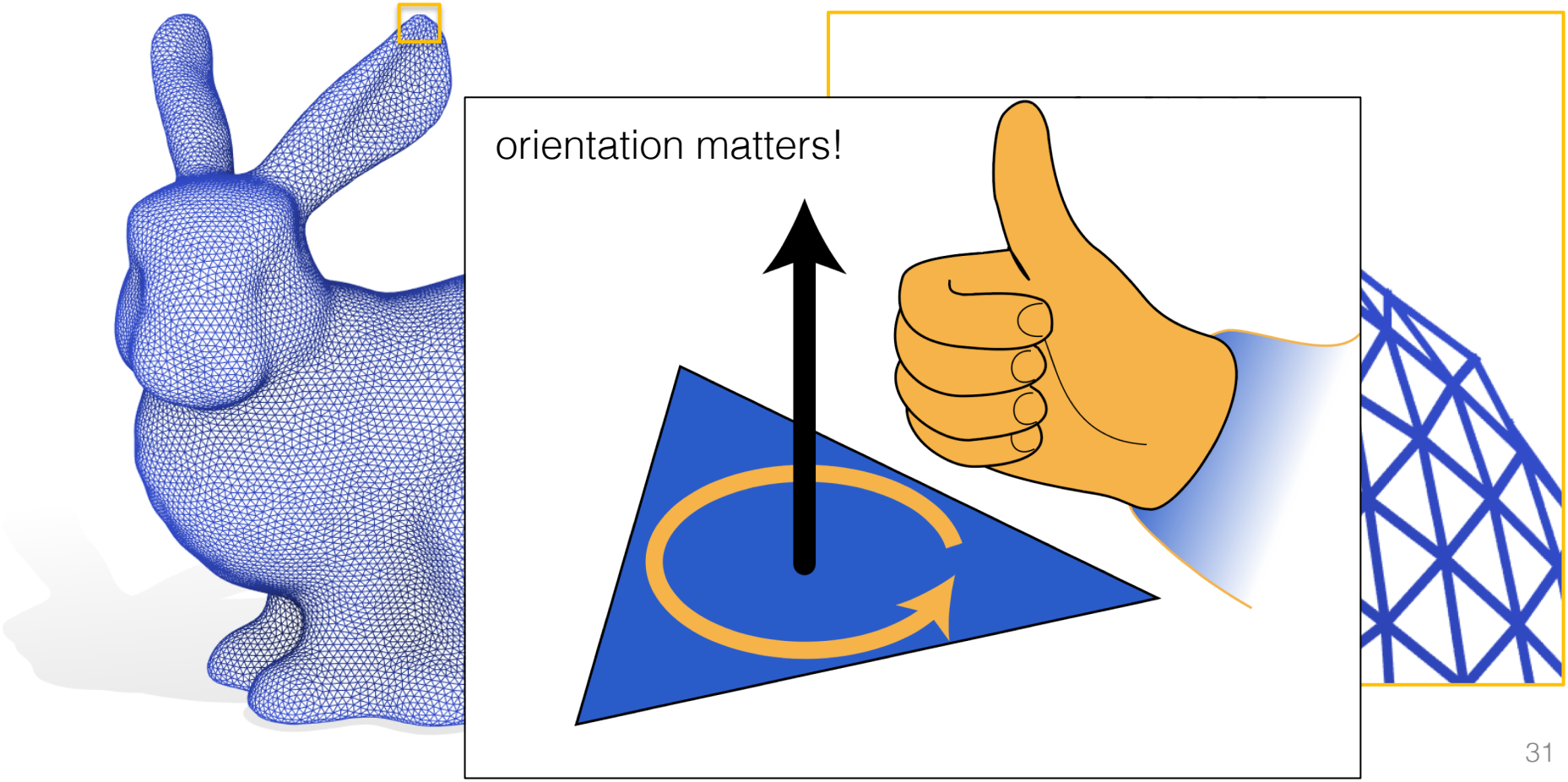


Triangle meshes discretize surfaces…

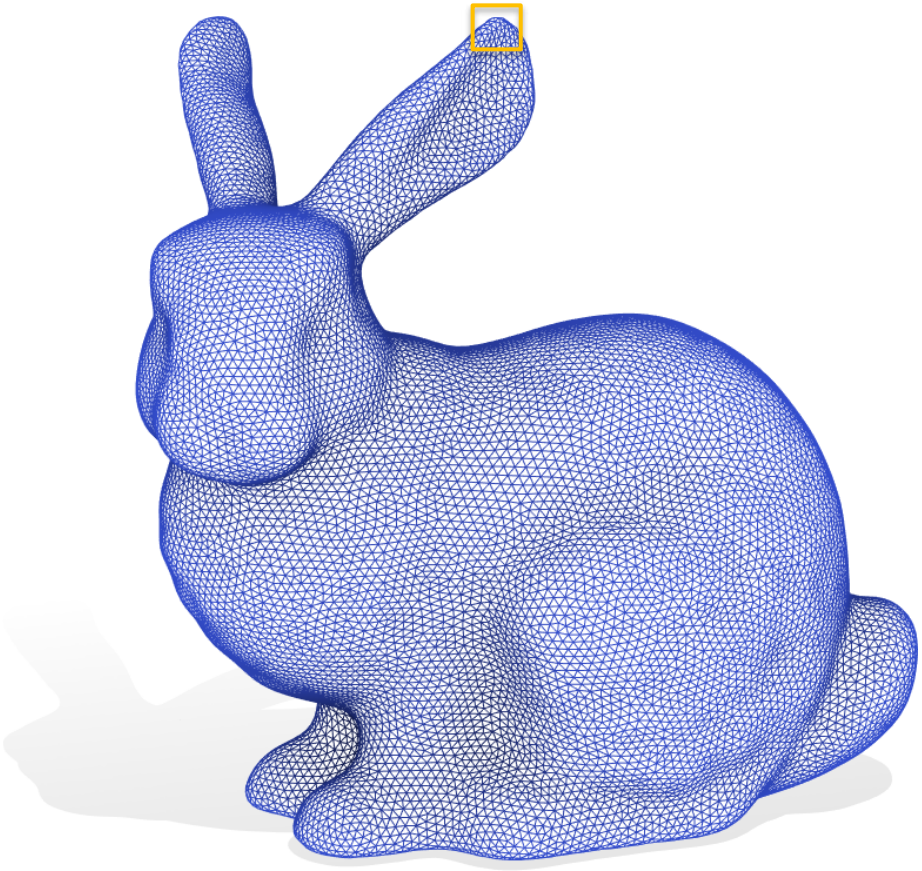


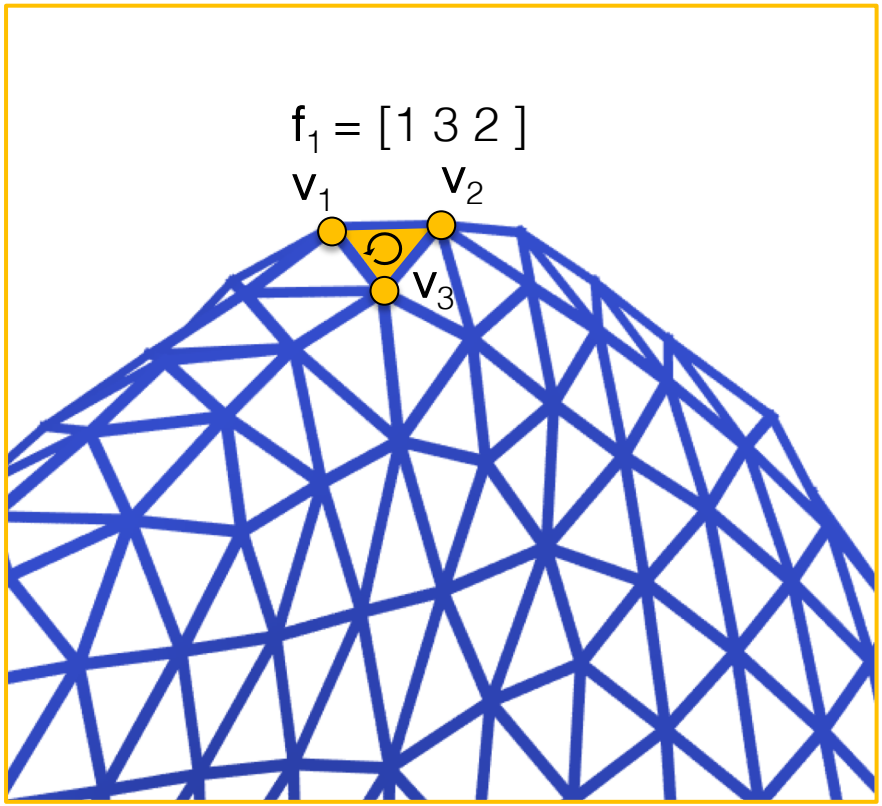


Triangle meshes discretize surfaces…

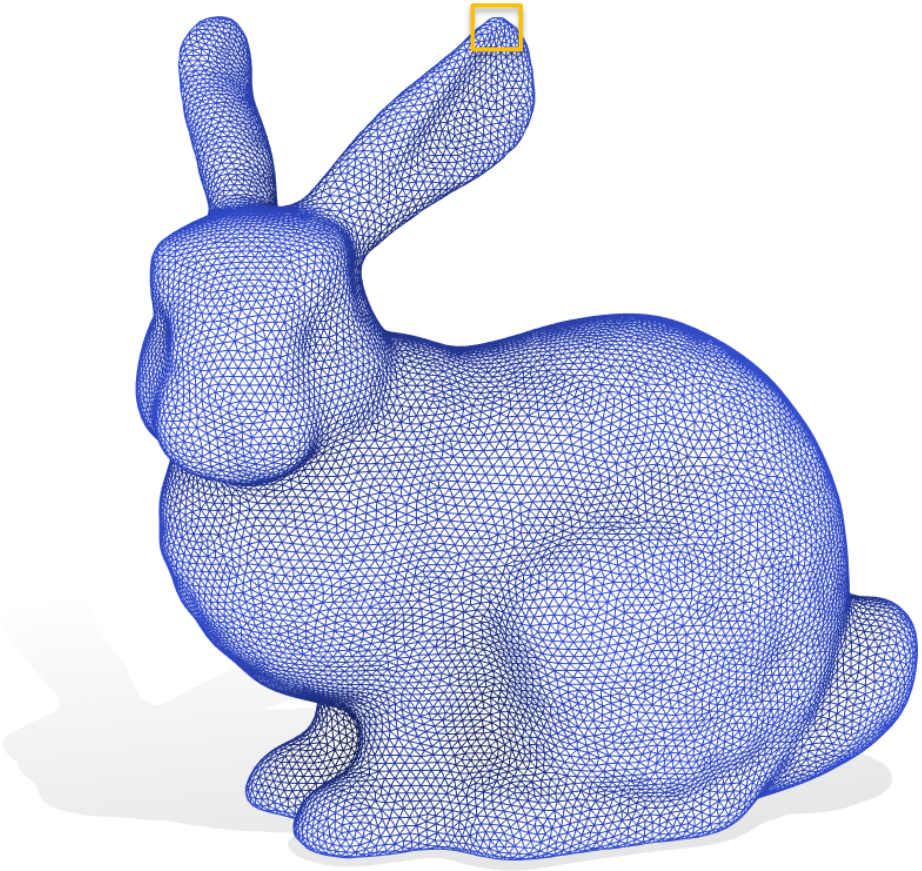


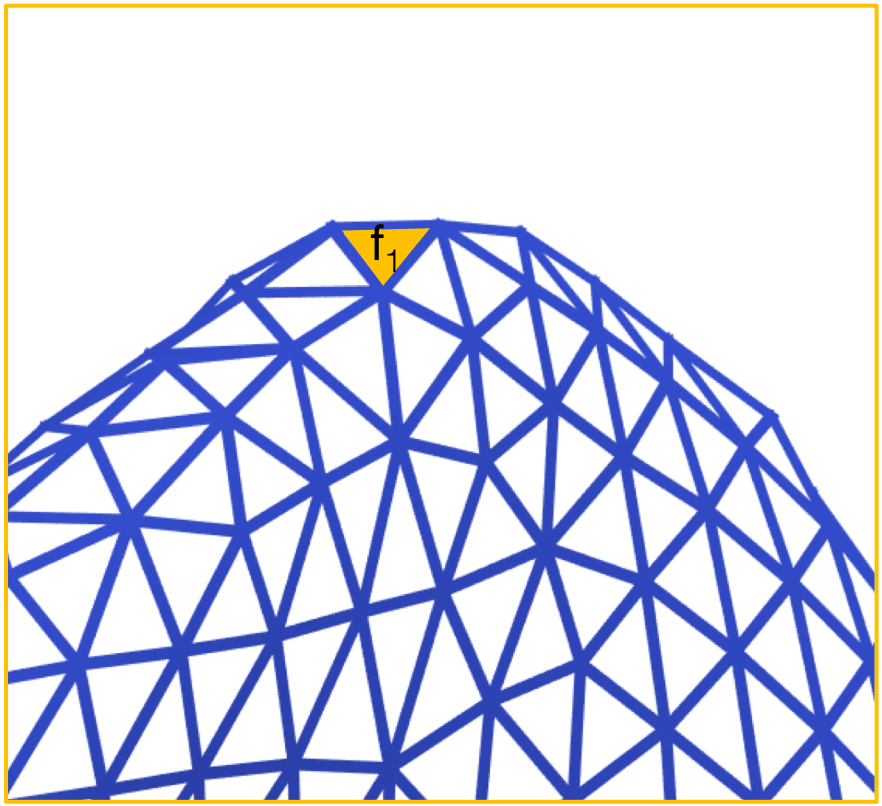
Triangle meshes discretize surfaces…



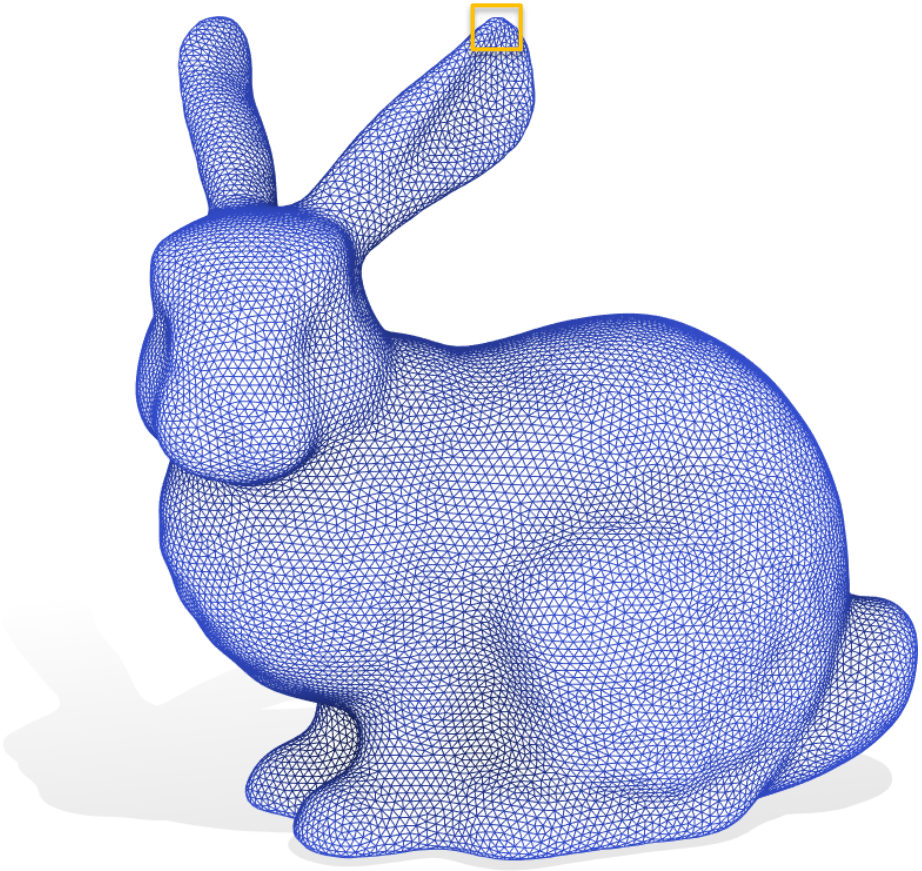


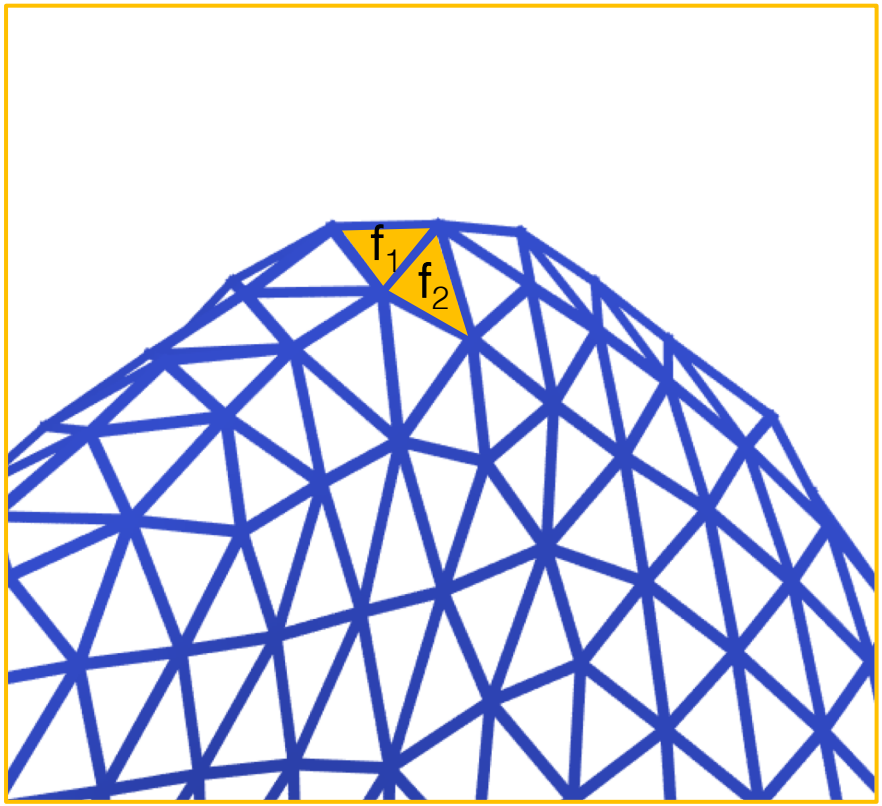
Triangle meshes discretize surfaces…



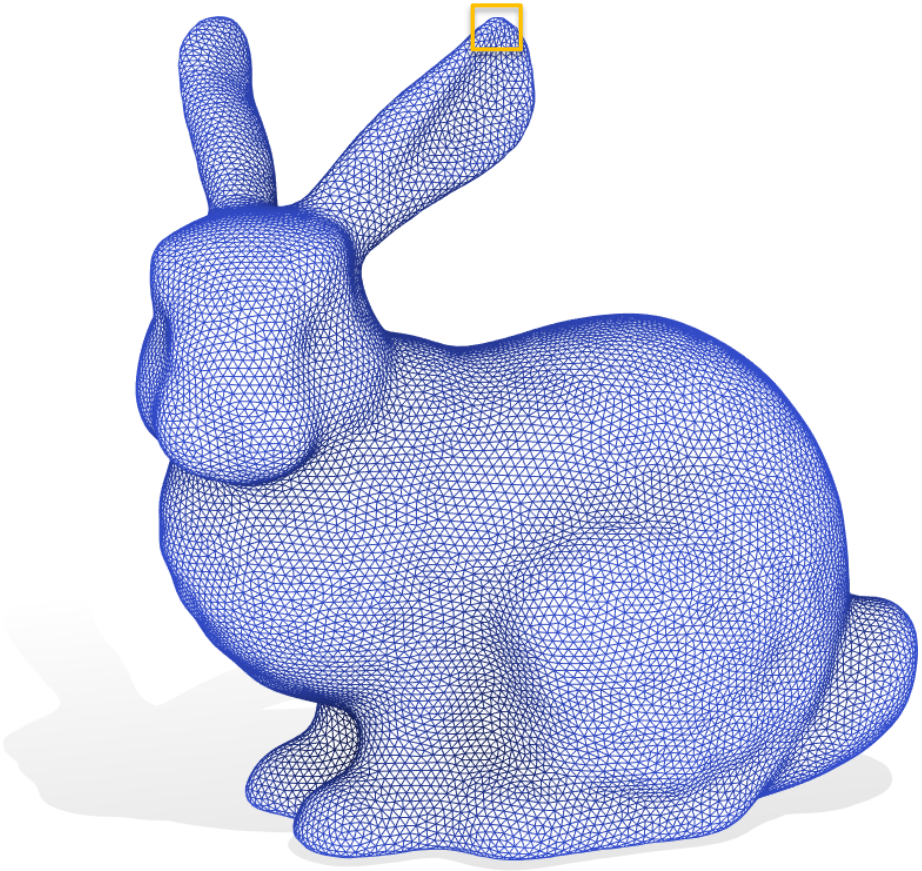


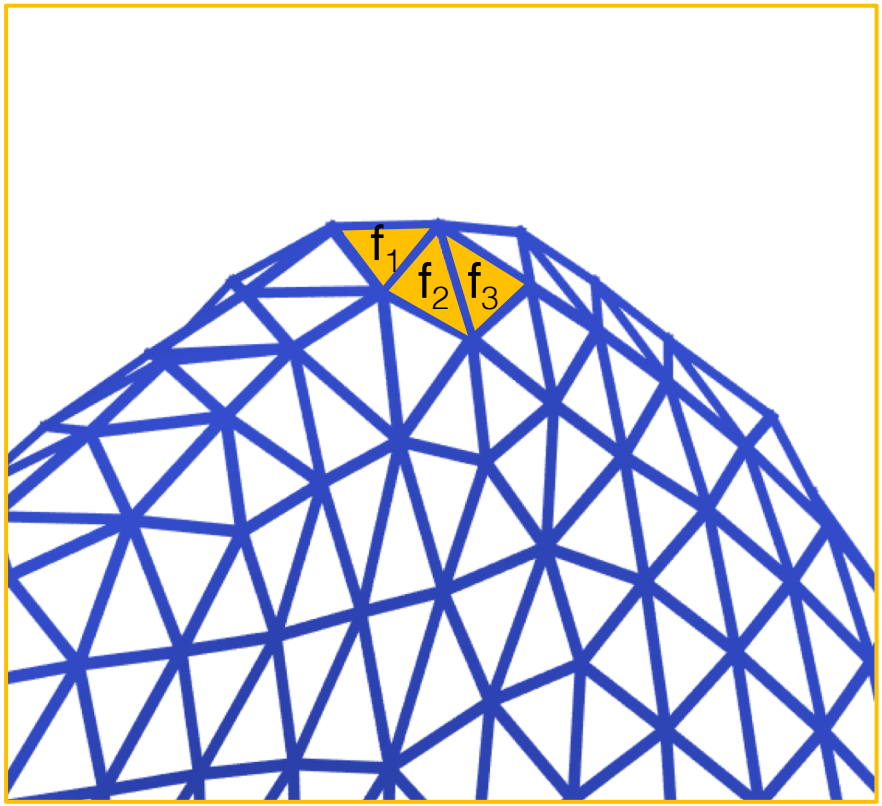
Triangle meshes discretize surfaces…



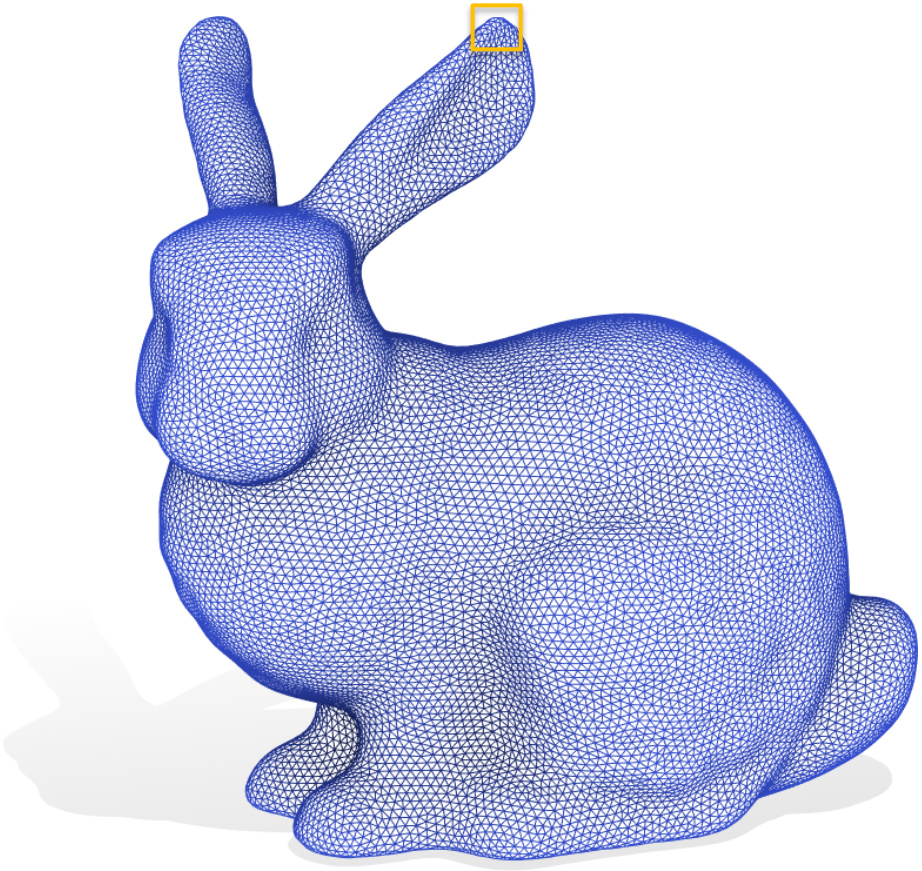


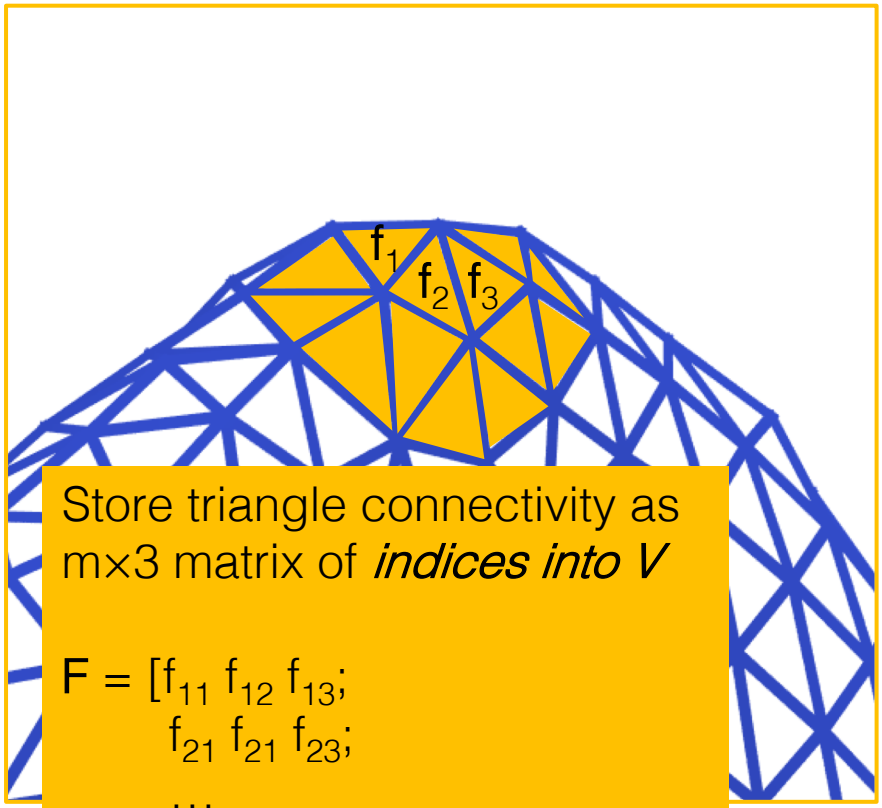
Triangle meshes discretize surfaces…





Triangle meshes discretize surfaces…

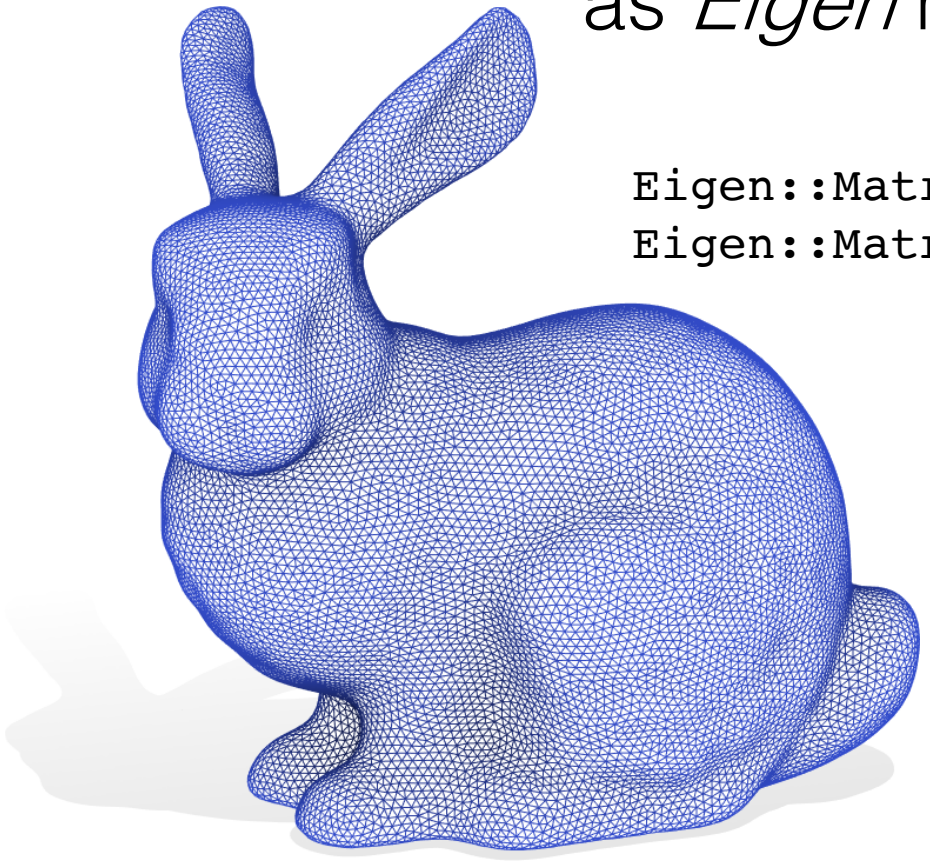




Store triangle connectivity as m×3 matrix of ***indices into V***

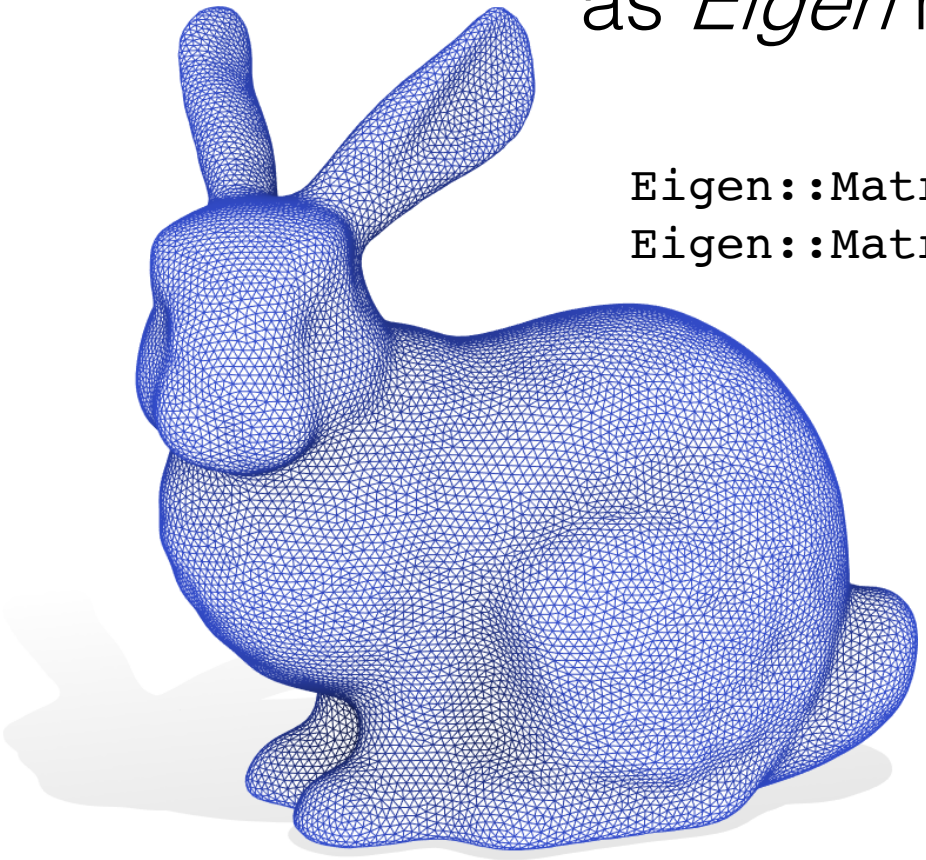
$$F = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{21} & f_{23} \\ \dots & & \\ f_{n1} & f_{n2} & f_{n3} \end{bmatrix}$$

Libigl expects vertex positions and triangle indices as *Eigen* matrices



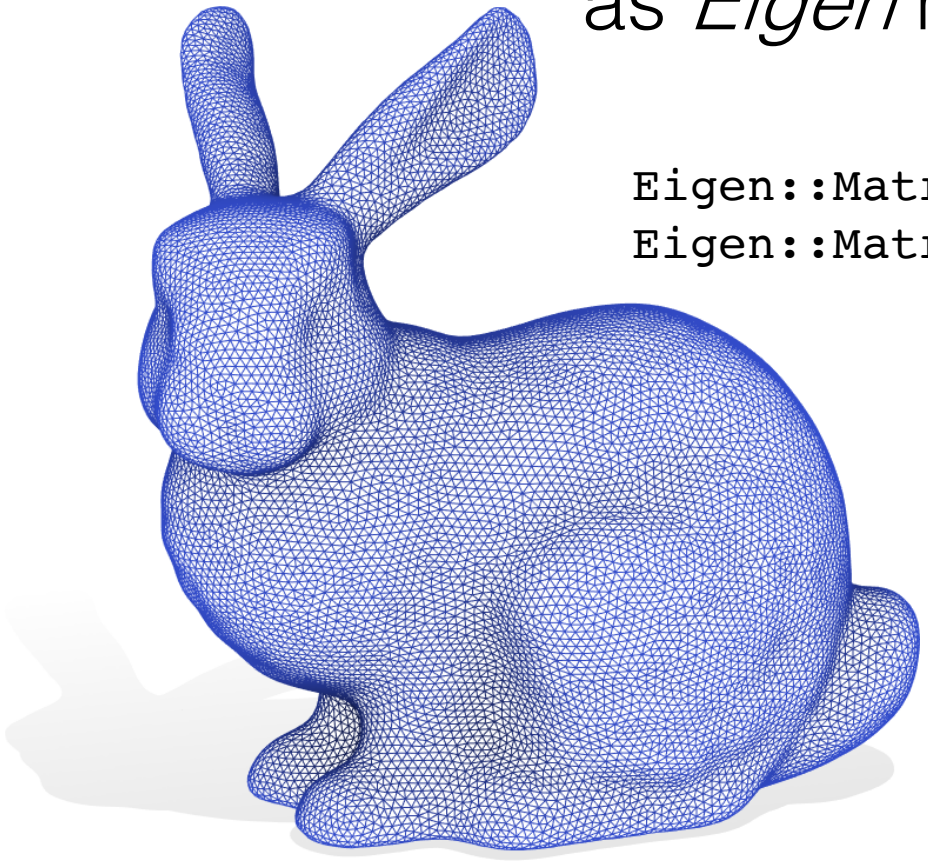
```
Eigen::Matrix<double,Eigen::Dynamic,3> V;
Eigen::Matrix<int,Eigen::Dynamic,3> F;
```

Libigl expects vertex positions and triangle indices as *Eigen* matrices



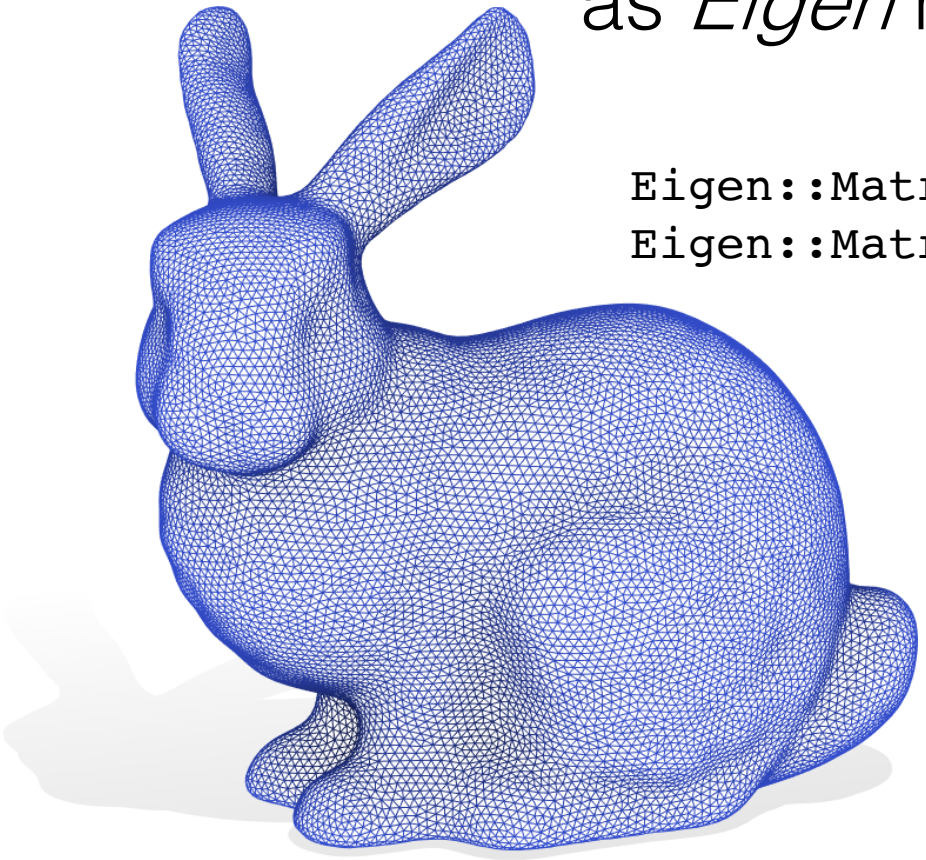
```
Eigen::Matrix<double,Eigen::Dynamic,3> V;
Eigen::Matrix<int,Eigen::Dynamic,3> F;
```

Libigl expects vertex positions and triangle indices as *Eigen* matrices



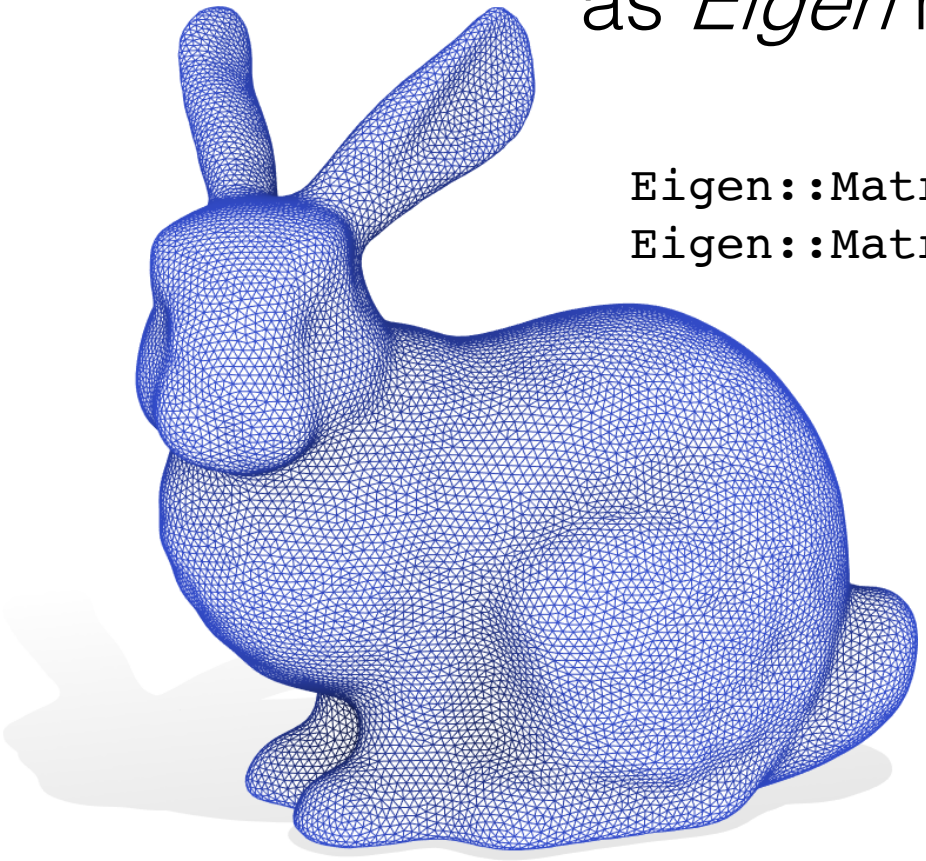
```
Eigen::Matrix<double,Eigen::Dynamic,3> V;
Eigen::Matrix<int,Eigen::Dynamic,3> F;
```

Libigl expects vertex positions and triangle indices as *Eigen* matrices



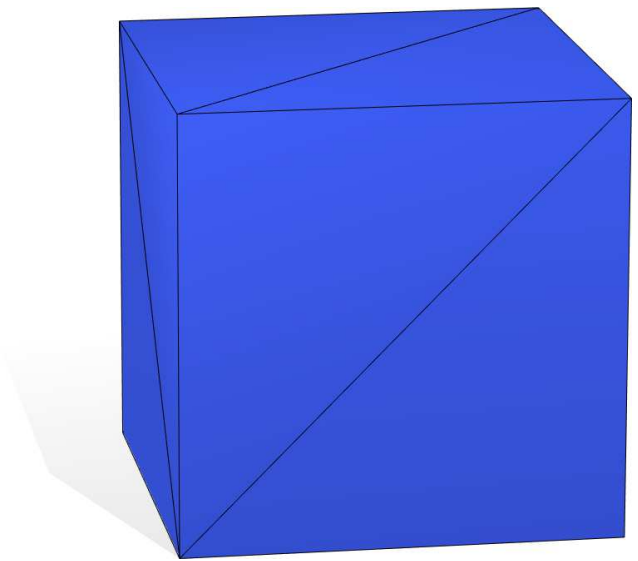
```
Eigen::Matrix<double,Eigen::Dynamic,3> V;
Eigen::Matrix<int,Eigen::Dynamic,3> F;
```

Libigl expects vertex positions and triangle indices as *Eigen* matrices



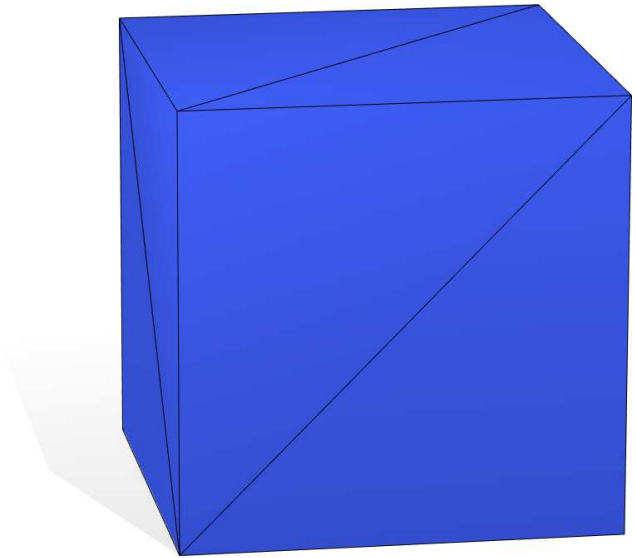
```
Eigen::MatrixXd V;
Eigen::MatrixXi F;
```

Libigl expects vertex positions and triangle indices as *Eigen* matrices



```
Eigen::MatrixXd V(8,3);
V<<
  0.0, 0.0, 0.0,
  0.0, 0.0, 1.0,
  0.0, 1.0, 0.0,
  0.0, 1.0, 1.0,
  1.0, 0.0, 0.0,
  1.0, 0.0, 1.0,
  1.0, 1.0, 0.0,
  1.0, 1.0, 1.0;
Eigen::MatrixXi F(12,3);
F<<
  0, 6, 4,
  0, 2, 6,
  0, 3, 2,
  0, 1, 3,
  2, 7, 6,
  2, 3, 7,
  4, 6, 7,
  4, 7, 5,
  0, 4, 5,
  0, 5, 1,
  1, 5, 7,
  1, 7, 3;
```

Libigl expects vertex positions and triangle indices as *Eigen* matrices



```
Eigen::MatrixXd V;
Eigen::MatrixXi F;
igl::read_triangle_mesh(“cube.obj”,V,F);
```

Libigl expects vertex positions and triangle indices as *Eigen* m


```
# cube.obj
v 0 0 0
v 0 0 1
v 0 1 0
v 0 1 1
v 1 0 0
v 1 0 1
v 1 1 0
v 1 1 1
f 1 7 5
f 1 3 7
f 1 4 3
f 1 2 4
f 3 8 7
f 3 4 8
f 5 7 8
f 5 8 6
f 1 5 6
f 1 6 2
f 2 6 8
f 2 8 4
```

```
Eigen:                    V;
Eigen:                    F;
igl::re                   le_mesh("cube.obj",V,F);
```

Raw matrices are …

- memory efficient and cache friendly,
- indices are (often) simpler to debug than pointers,
- trivially copied and serialized,
- immediately passable to other libraries:
  OpenGL, OpenCV, MATLAB, embree, …

Vertex position matrices immediately *afford* linear algebra

$$\mathbf{u} = \mathbf{v} - \lambda \Delta \mathbf{v}$$

Vertex position matrices immediately *afford* linear algebra

$$\mathbf{u} = \mathbf{v} - \lambda \Delta \mathbf{v}$$

```
Eigen::MatrixXd U = V - lamdba*L*V;
```

Vertex position matrices immediately *afford* linear algebra

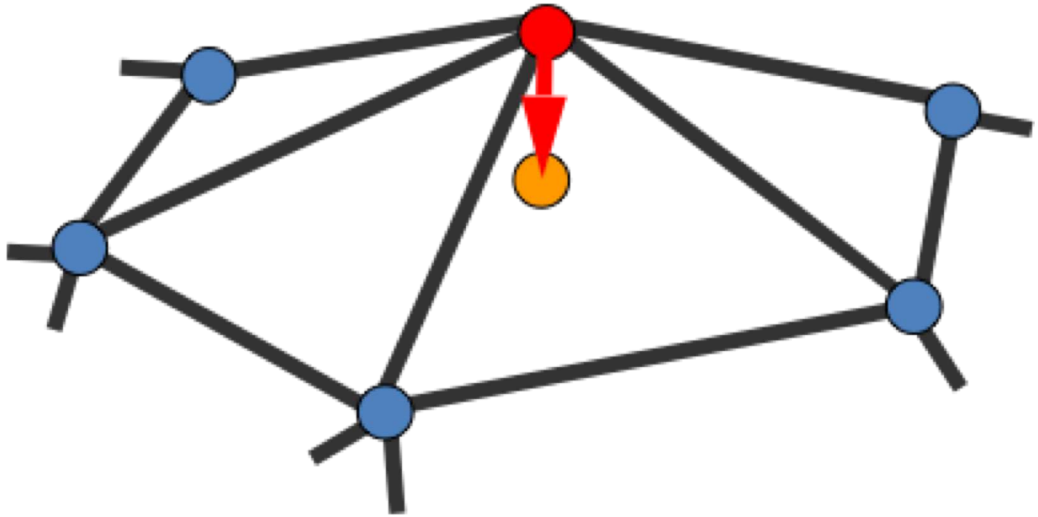
$$\mathbf{u} = \mathbf{v} - \lambda \Delta \mathbf{v}$$

```
Eigen::MatrixXd U = V - lamdba*L*V;
```

```
Eigen::SparseMatrix<double> L;
```

Vertex position matrices immediately *afford* linear algebra

$$\mathbf{u}_i = \mathbf{v}_i - \lambda(\Delta\mathbf{v})_i$$



Vertex position matrices immediately *afford* linear algebra

$$\mathbf{u}_i = \mathbf{v}_i - \lambda(\Delta\mathbf{v})_i$$

$$\mathbf{u}_i = \mathbf{v}_i - \lambda \underbrace{\left(\sum_{j \in N(i)} w_{ij}(\mathbf{v}_i - \mathbf{v}_j) \right)}_{\text{linear in } \mathbf{V}}$$

Vertex position matrices immediately *afford* linear algebra

$$\mathbf{u}_i = \mathbf{v}_i - \lambda(\Delta\mathbf{v})_i$$

$$\mathbf{u}_i = \mathbf{v}_i - \lambda \underbrace{\left(\sum_{j \in N(i)} w_{ij}(\mathbf{v}_i - \mathbf{v}_j) \right)}_{\text{linear in } \mathbf{V}}$$

$$\mathbf{U} = \mathbf{V} - \lambda \mathbf{L}\,\mathbf{V}$$

Vertex position matrices immediately *afford* linear algebra

$$\mathbf{u}_i = \mathbf{v}_i - \lambda(\Delta\mathbf{v})_i$$

$$\mathbf{u}_i = \mathbf{v}_i - \lambda \underbrace{\left(\sum_{j \in N(i)} w_{ij}(\mathbf{v}_i - \mathbf{v}_j) \right)}_{\text{linear in } \mathbf{V}}$$

$$\mathbf{U} = \mathbf{V} - \lambda \mathbf{L}\mathbf{V}$$

acts on each column (coordinate) in $\mathbf{V}$

Vertex position matrices immediately *afford* linear algebra

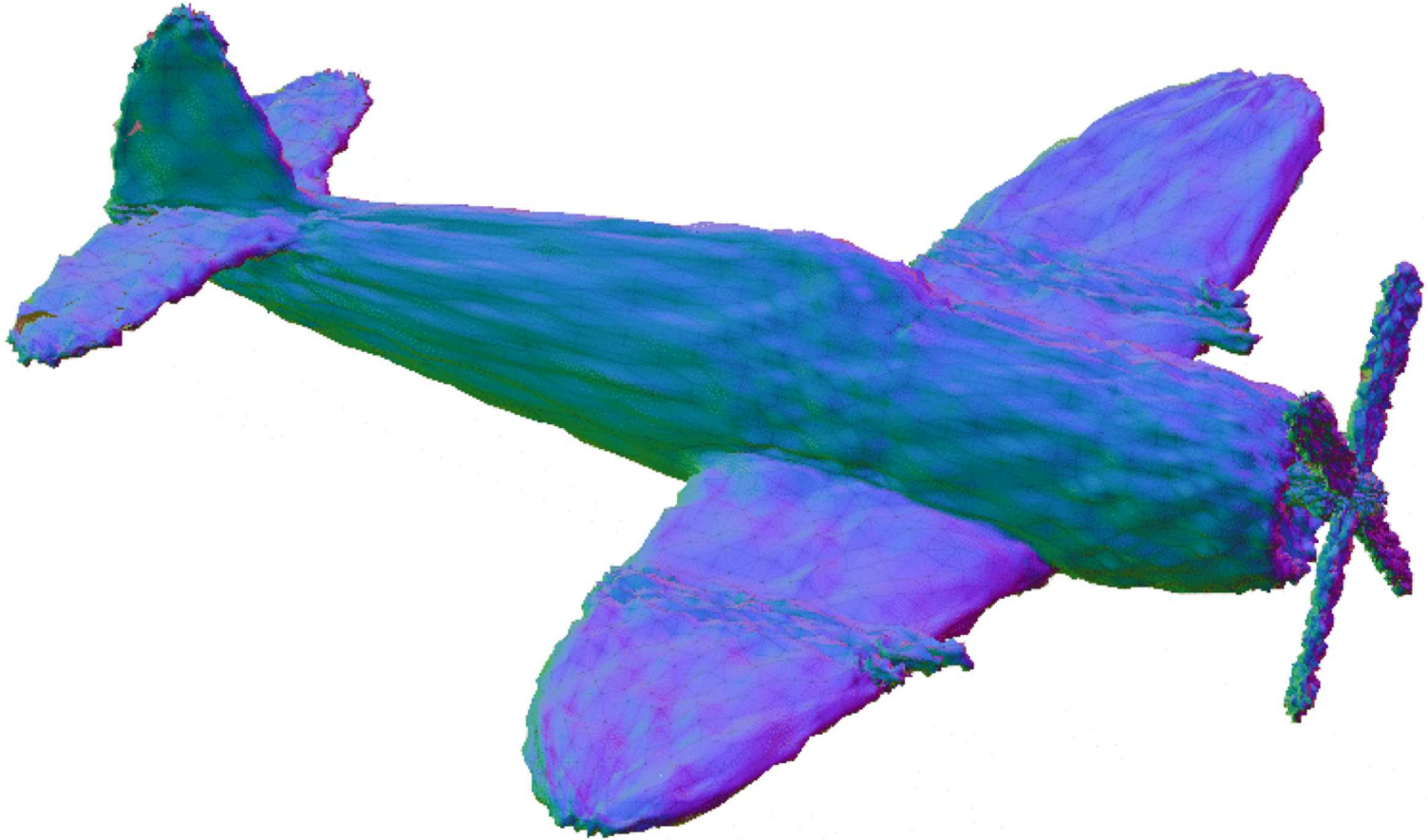
$$\mathbf{u}_i = \mathbf{v}_i - \lambda(\Delta\mathbf{v})_i$$

$$\mathbf{u}_i = \mathbf{v}_i - \lambda \underbrace{\left(\sum_{j \in N(i)} w_{ij}(\mathbf{v}_i - \mathbf{v}_j) \right)}_{\text{linear in } \mathbf{V}}$$

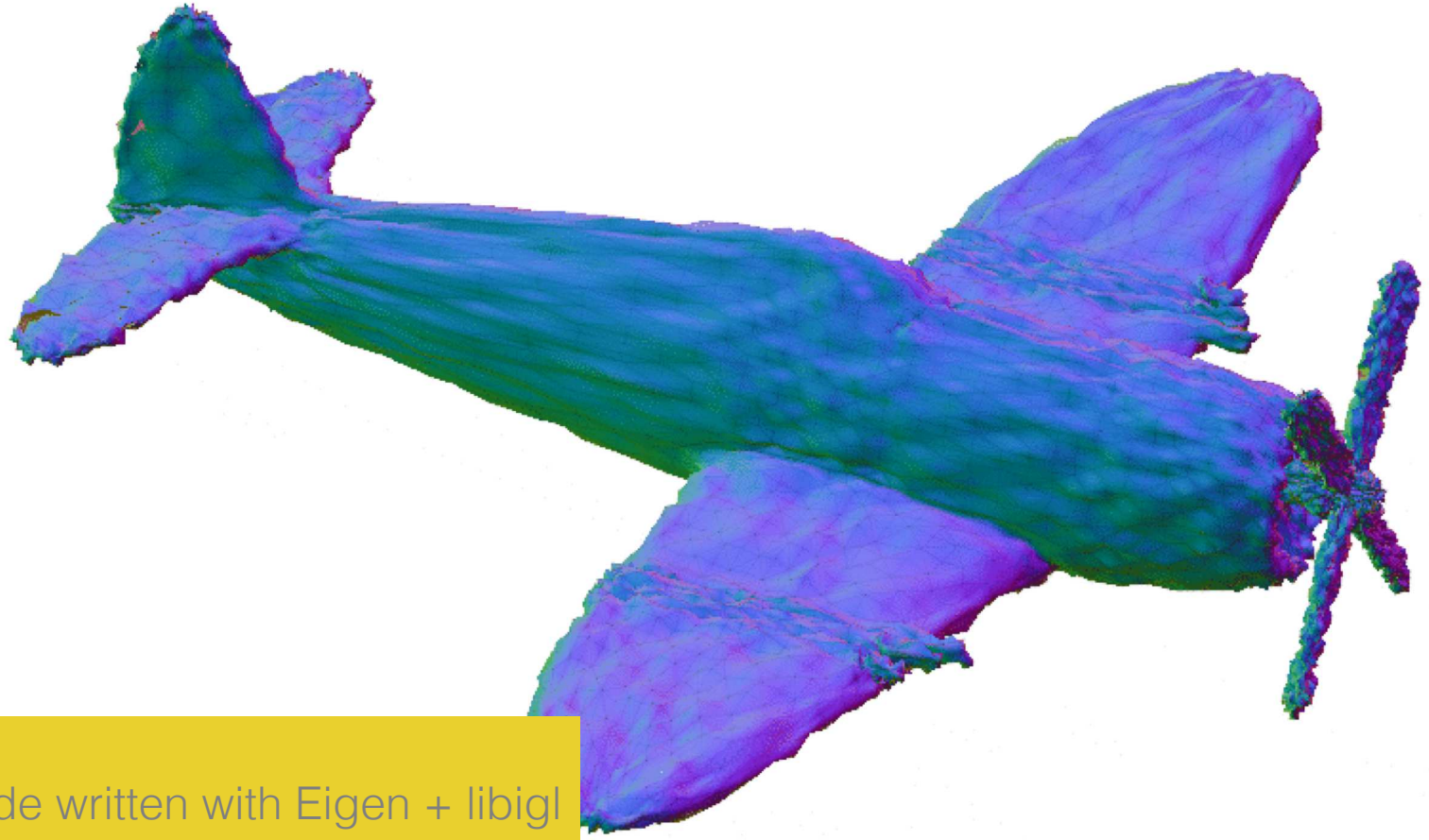
$$\mathbf{U} = \mathbf{V} - \lambda \mathbf{L}\,\mathbf{V}$$

math → code

```
Eigen::MatrixXd U = V - lamdba*L*V;
```



```
Eigen::MatrixXd U = V - lamdba*L*V;
```



C++ code written with Eigen + libigl
looks a lot like MATLAB or Python

```
U = V - lamdba*L*V;
```

Other Matlab-style functions

Libigl implements a variety of other routines with the same api and functionality as common Matlab functions.

Name	Description
`igl::any_of`	Whether any elements are non-zero (true)
`igl::cat`	Concatenate two matrices (especially useful for dealing with Eigen sparse matrices)
`igl::ceil`	Round entries up to nearest integer
`igl::cumsum`	Cumulative sum of matrix elements
`igl::colon`	Act like Matlab's `:` , similar to Eigen's `LinSpaced`
`igl::cross`	Cross product per-row
`igl::dot`	dot product per-row
`igl::find`	Find subscripts of non-zero entries
`igl::floor`	Round entries down to nearest integer
`igl::histc`	Counting occurrences for building a histogram
`igl::hsv_to_rgb`	Convert HSV colors to RGB (cf. Matlab's `hsv2rgb`)
`igl::intersect`	Set intersection of matrix elements.
`igl::isdiag`	Determine whether matrix is diagonal
`igl::jet`	Quantized colors along the rainbow.
`igl::median`	Compute the median per column
`igl::mode`	Compute the mode per column
`igl::null`	Compute the null space basis of a matrix
`igl::nchoosek`	Compute all k-size combinations of n-long vector
`igl::orth`	Orthogonalization of a basis
`igl::parula`	Generate a quantized colormap from blue to yellow
`igl::randperm`	Generate a random permutation of [0,…,n–1]
`igl::rgb_to_hsv`	Convert RGB colors to HSV (cf. Matlab's `rgb2hsv`)
`igl::setdiff`	Set difference of matrix elements
`igl::sort`	Sort elements or rows of matrix
`igl::speye`	Identity as sparse matrix
`igl::sum`	Sum along columns or rows (of sparse matrix)
`igl::unique`	Extract unique elements or rows of matrix

libigl embraces matrices as primary mesh data structure

input

```
Eigen::MatrixXd V; // #V by 3
Eigen::MatrixXi F; // #F by 3
```

libigl embraces matrices as primary mesh data structure

input

```
Eigen::MatrixXd V; // #V by 3
Eigen::MatrixXi F; // #F by 3
```

output

```
Eigen::VectorXd X; // #V by 1
```

libigl embraces matrices as primary mesh data structure

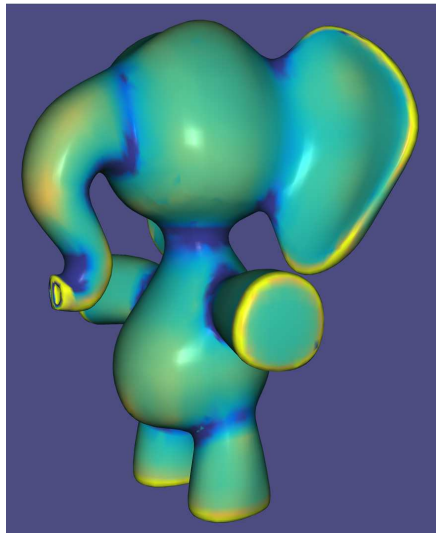
input

```
Eigen::MatrixXd V; // #V by 3
Eigen::MatrixXi F; // #F by 3
```

output

```
Eigen::VectorXd X; // #V by 1
```

```
igl::gaussian_curvature(V,F,X);
```



libigl embraces matrices as primary mesh data structure

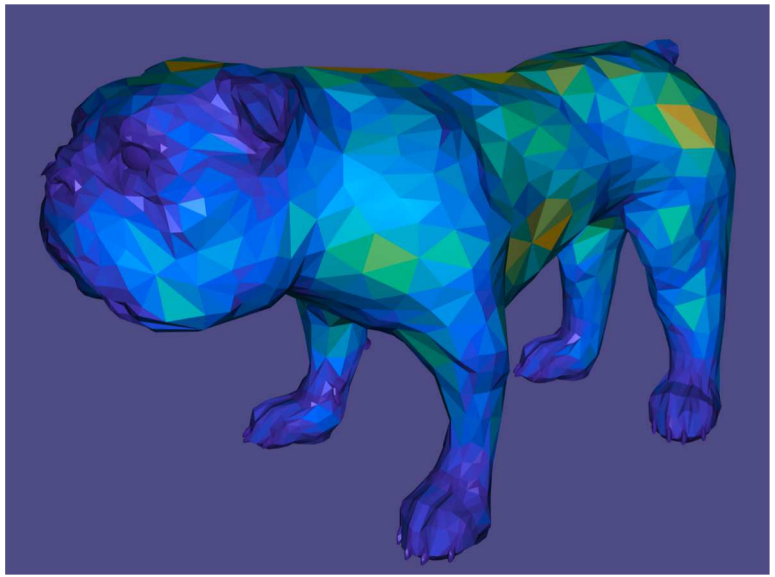
input

```
Eigen::MatrixXd V; // #V by 3
Eigen::MatrixXi F; // #F by 3
```

```
igl::doublearea(V,F,X);
```

output

```
Eigen::VectorXd X; // #F by 1
```



libigl embraces matrices as primary mesh data structure

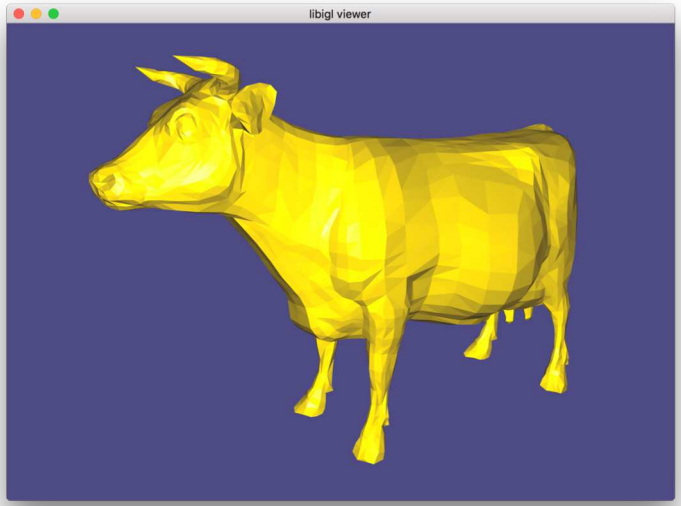
input

output

```
Eigen::MatrixXd V; // #V by 3
Eigen::MatrixXi F; // #F by 3
```

```
Eigen::MatrixXd X; // #F by 3
```

```
igl::per_face_normals(V,F,X);
```



libigl embraces matrices as primary mesh data structure

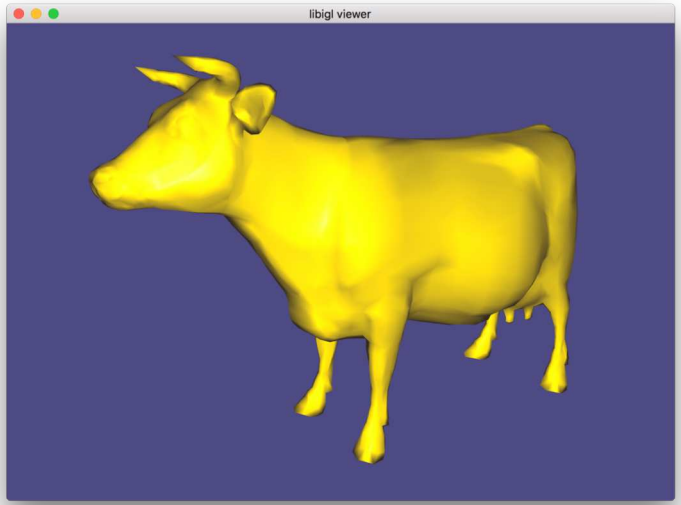
input

output

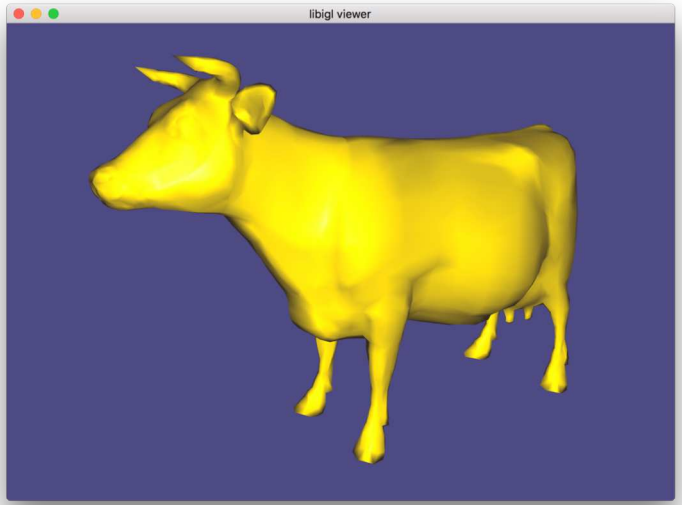
```
Eigen::MatrixXd V; // #V by 3
Eigen::MatrixXi F; // #F by 3
```

```
Eigen::MatrixXd X; // #V by 3
```

```
igl::per_vertex_normals(V,F,X);
```



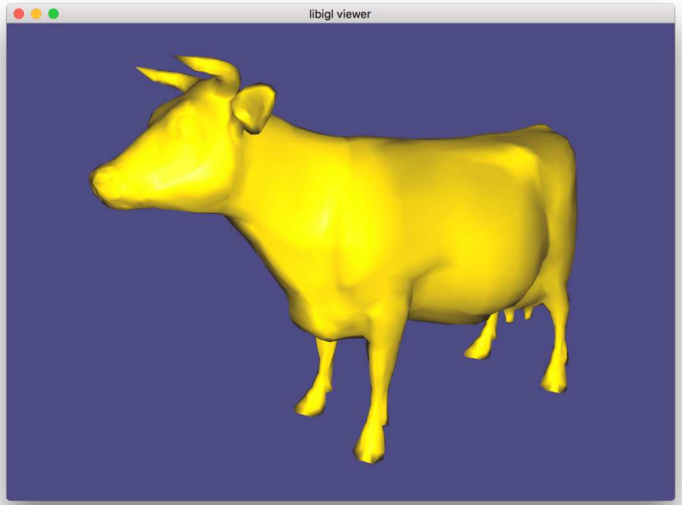
Wait! What about half-edge data-structures? How do I loop over the one-ring?



Wait! What about half-edge data-structures? How do I loop over the one-ring?

Standard loop over vertices

```
for vertex i
    N(i) = [0,0,0];
    for face f adjacent to i
        N(i) += f's normal
    N(i).normalize
```

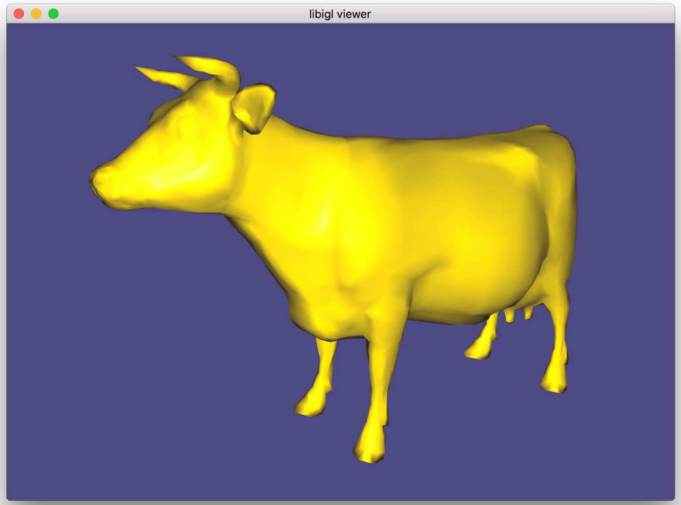


Wait! What about half-edge data-structures? How do I loop over the one-ring?

Standard loop over vertices

```
for vertex i
    N(i) = [0,0,0];
    for face f adjacent to i
        N(i) += f's normal
    N(i).normalize
```

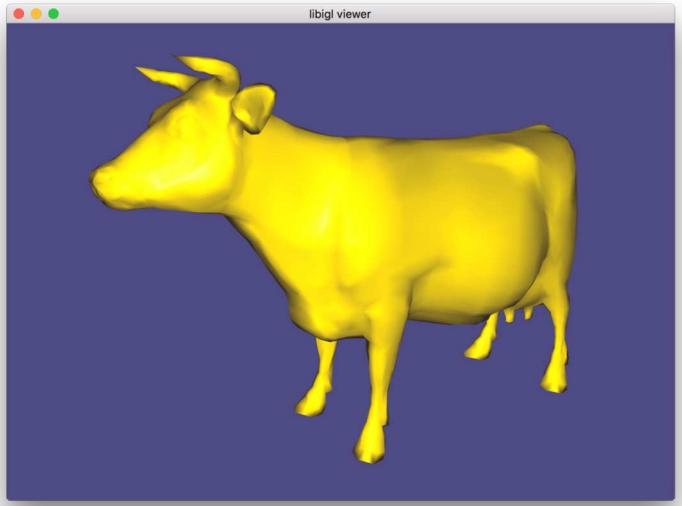
Gather values in place



Wait! What about half-edge data-structures? How do I loop over the one-ring?

Loop over faces

```
N(:,:) = 0;
for face f
   for corner vertex i
      N(i) += f's normal
for vertex i
   N(i).normalize
```

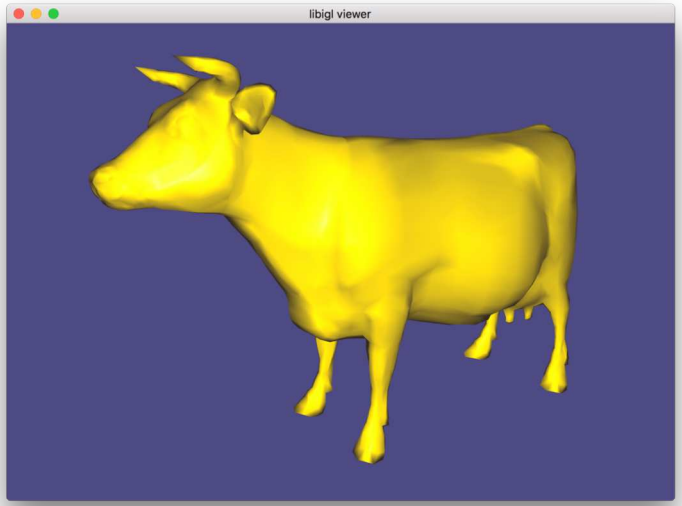


Wait! What about half-edge data-structures? How do I loop over the one-ring?

Loop over faces

```
N(:,:) = 0;
for face f
    for corner vertex i
        N(i) += f's normal
for vertex i
    N(i).normalize
```

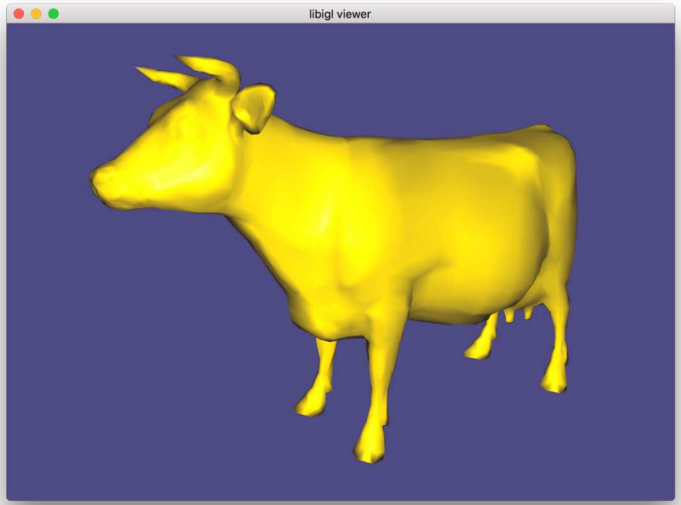
Throw values and accumulate



Wait! What about half-edge data-structures? How do I loop over the one-ring?

Loop over faces

```
N = MatrixXd::Zero(n,3);
for(int f=0; f<F.rows(); f++)
   for(int c=0; c<3; c++)
      N.row(F(f,c)) += NF.row(f);
for(int i=0; i<V.rows(); i++)
   N.row(i).normalize();
```



We founded libigl on four core principles

1. Simple, matrix-based API

input

```
Eigen::MatrixXd V; // #V by 3
Eigen::MatrixXi F; // #F by 3
```

output

```
Eigen::MatrixXd X; // #V by 3
```

```
igl::per_vertex_normals(V,F,X);
```

We founded libigl on four core principles

1. Simple, matrix-based API
2. Header-only “installation”

```
#include <igl/read_triangle_mesh.h>
#include <igl/per_vertex_normals.h>
int main()
{
  igl::read_triangle_mesh(“cow.obj”,V,F);
  igl::per_vertex_normals(V,F,N);
}
```

We founded libigl on four core principles

1. Simple, matrix-based API
2. Header-only “installation”
3. One function One file

```
// libigl/include/igl/per_vertex_normals.h
#ifndef IGL_PER_VERTEX_H
#define IGL_PER_VERTEX_H
namespace igl
{
  void per_vertex_normals(
    const Eigen::MatrixXd & V,
    const Eigen::MatrixXd & F,
    Eigen::MatrixXd & N);
}
```

We founded libigl on four core principles

1. Simple, matrix-based API
2. Header-only “installation”
3. One function One file
4. Zero dependencies

We founded libigl on four core principles

1. Simple, matrix-based API
2. Header-only “installation”
3. One function One file
4. ~~Zero~~ Minimal dependencies

```
libigl/include/igl/*.h
```

```
e.g., igl::per_vertex_normals()
```

depends on stl and Eigen

We founded libigl on four core principles

1. Simple, matrix-based API
2. Header-only “installation”
3. One function One file
4. ~~Zero~~ Minimal dependencies

`libigl/include/igl/*.h`

~70% of libigl

depends on stl and Eigen

We founded libigl on four core principles

1. Simple, matrix-based API
2. Header-only “installation”
3. One function One file
4. ~~Zero~~ Minimal dependencies

`libigl/include/igl/`**`tetgen`**`/*.h`

`e.g., igl`**`::tetgen::`**`tetrahedralize()`

depends on stl and Eigen, ***and Tetgen***

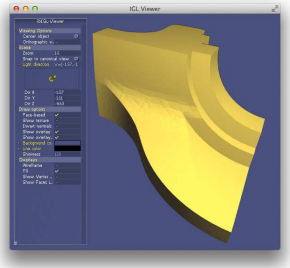
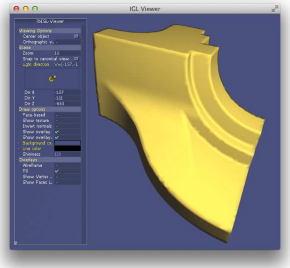
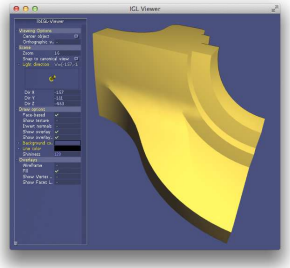
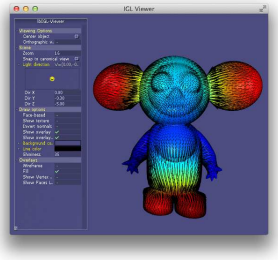
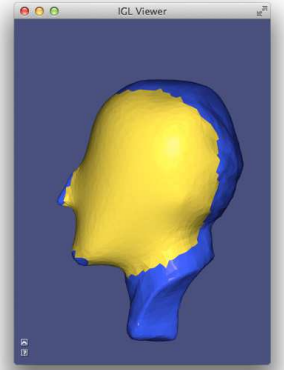
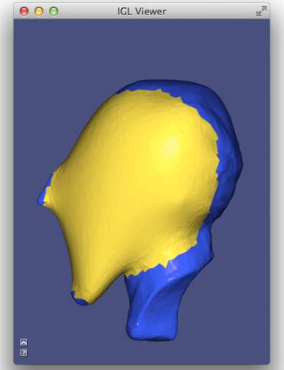
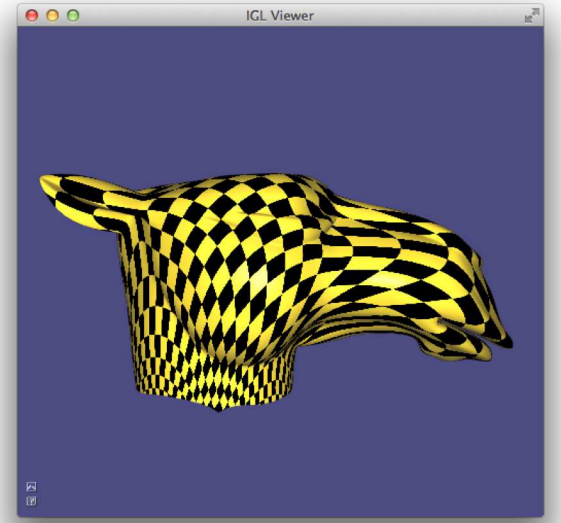
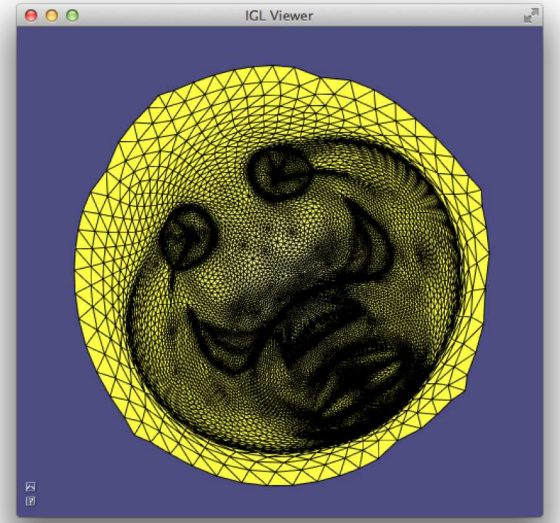
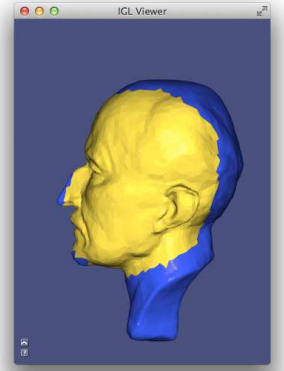
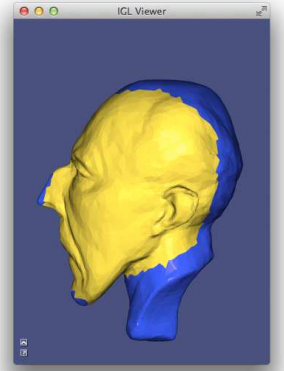
We founded libigl on four core principles

1. Simple, matrix-based API
2. Header-only “installation”
3. One function One file
4. ~~Zero~~ Minimal dependencies

`libigl/include/igl/`**`copyleft`**`/*.h`

`e.g., igl::`**`copyleft`**`::marching_cubes()`

GPL-type license, ***warning*** *for commerical users*

libigl tour of core functionalities

Tour of Core Functionalities

- https://github.com/libigl/libigl-course

  - Tutorials compilation with CMAKE
  - Differential Quantities
  - Parameterization
  - Deformation (and Locally Injective Maps)

Get started with:

```
git clone --recursive https://github.com/libigl/libigl.git
```

libigl Getting Up and Running with libigl

Getting Up and Running with libigl

- The core of libigl is header only
- The best way to start a new project is to use the basic CMAKE project
- Useful convenience tools
  - Render on memory buffer (screenshots)
  - Picking
  - Menu bar (nanogui)

libigl hot new features

New Features

- Serialization
- Booleans
- Python

Libigl Coding Tips (aka “How to code a SIGGRAPH project”)

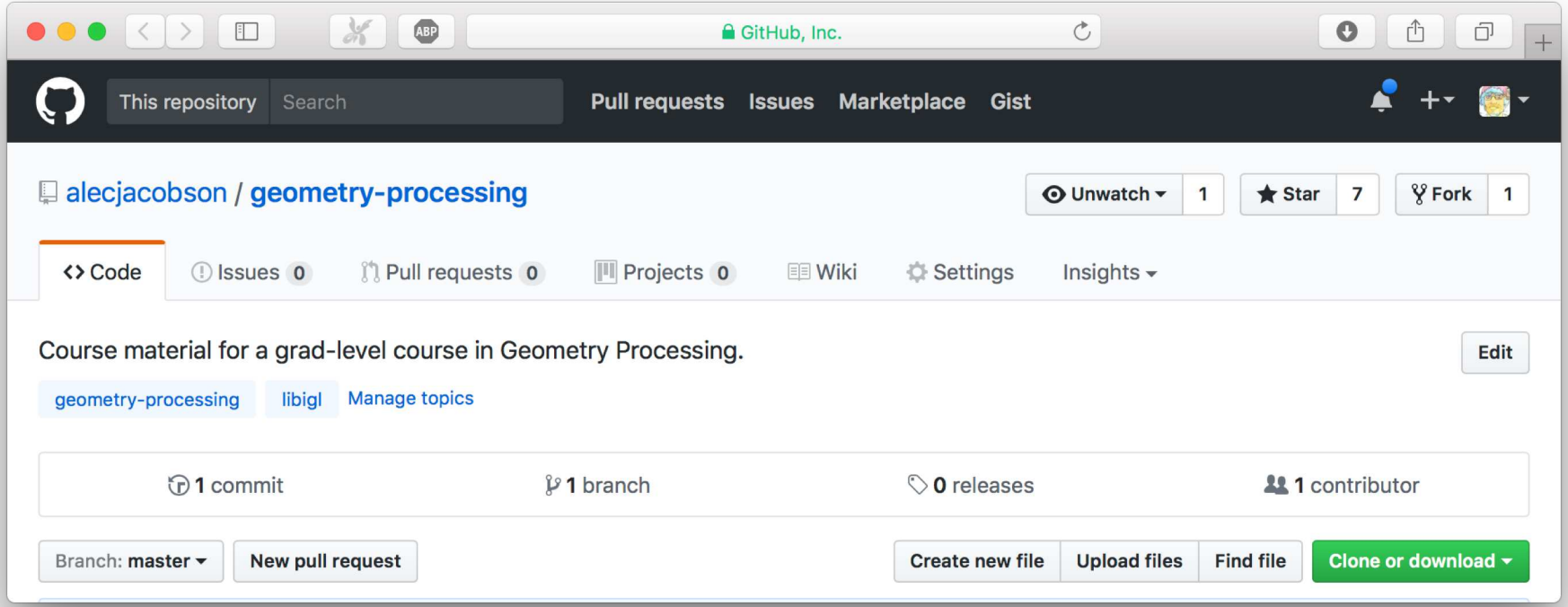
This is a short list of coding tips that will greatly reduce your pain and suffering before (and after) the SIGGRAPH deadline.

1. Serialize it all

The entire state of your application should be serializable, i.e. It should be possible to save it into a binary file and reload it at any point. This drastically simplifies debugging, since you can serialize just before a crash happens and debug from that point without running your complete algorithm again. Serializing all results shown in the paper’s figures enables quicker editing iterations before (and after) the deadline. It also allows you to share your results with others that wish to compare with your method. An additional tip is to serialize the state of the application on the window close event and automatically reload it when you launch it again.

2. Always assert

libigl coding tips



libigl open-source grad course in geometry processing

Another geometry processing course using libigl

https://github.com/danielepanozzo/gp

libigl Prototyping Geometry Processing Research in C++

Alec Jacobson
University of Toronto

https://github.com/libigl/libigl-course

Daniele Panozzo
New York University