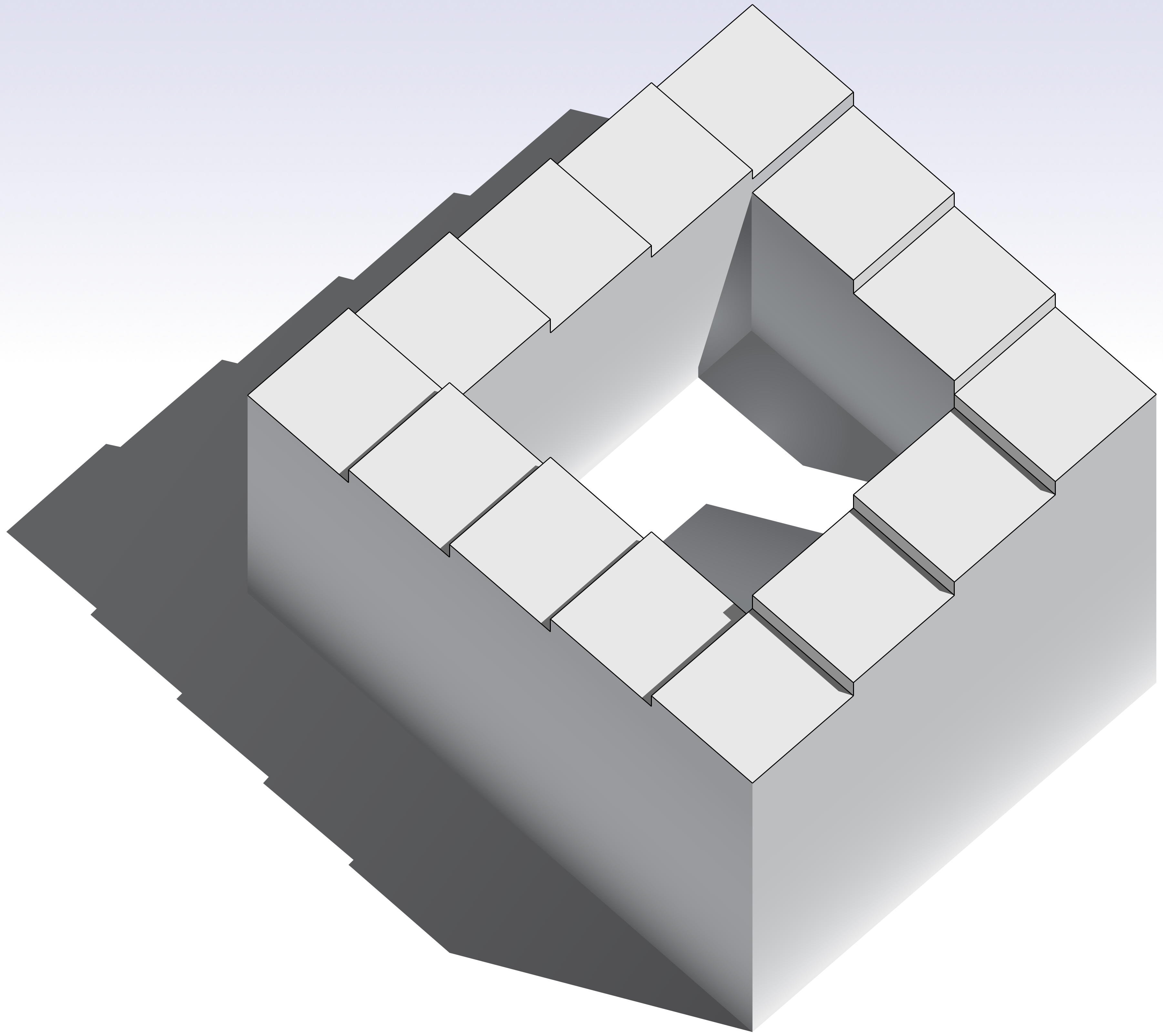
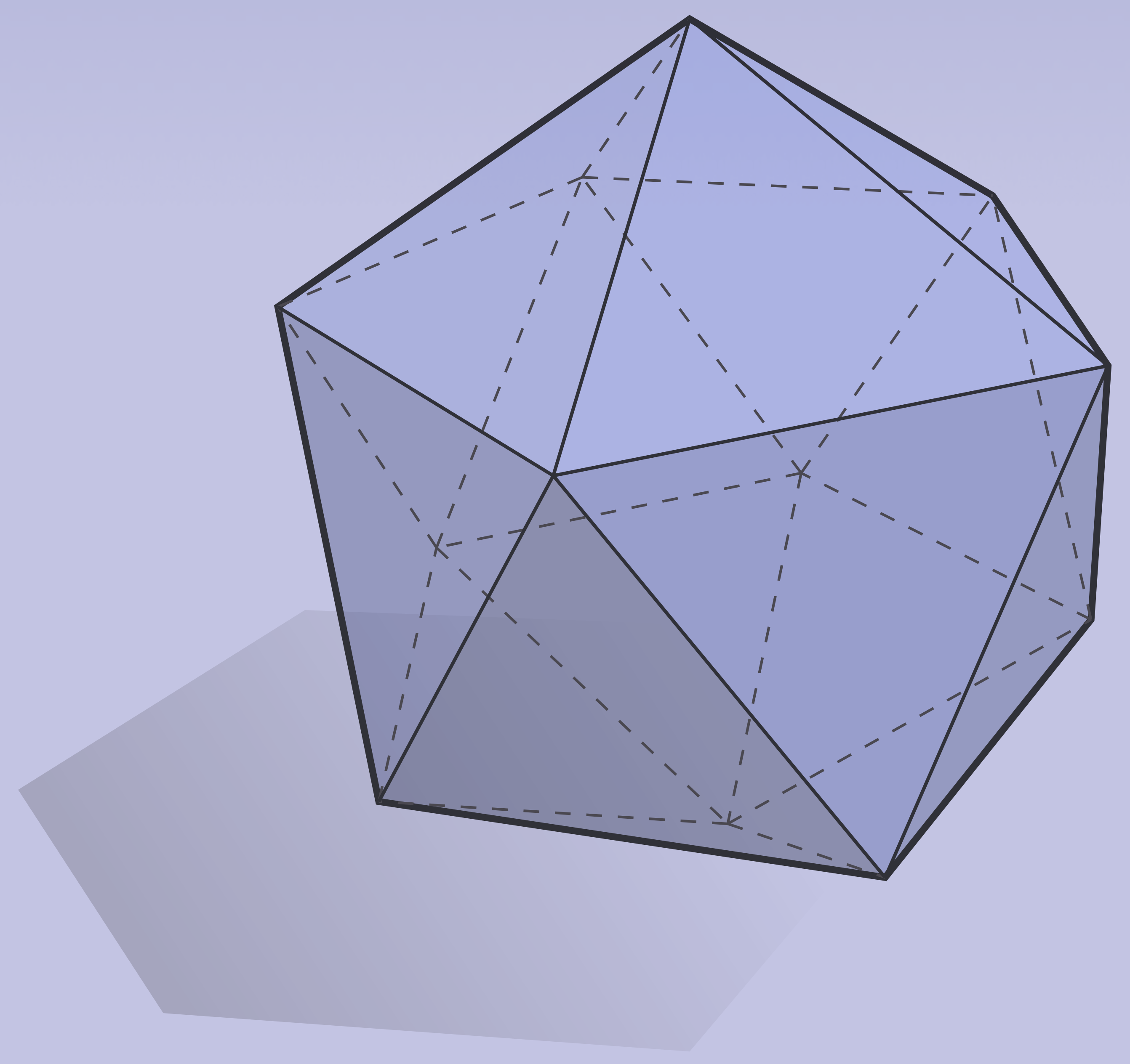


AMS Short Course

Discrete Differential Geometry

Joint Mathematics Meeting • San Diego, CA • January 2018

Demo Session

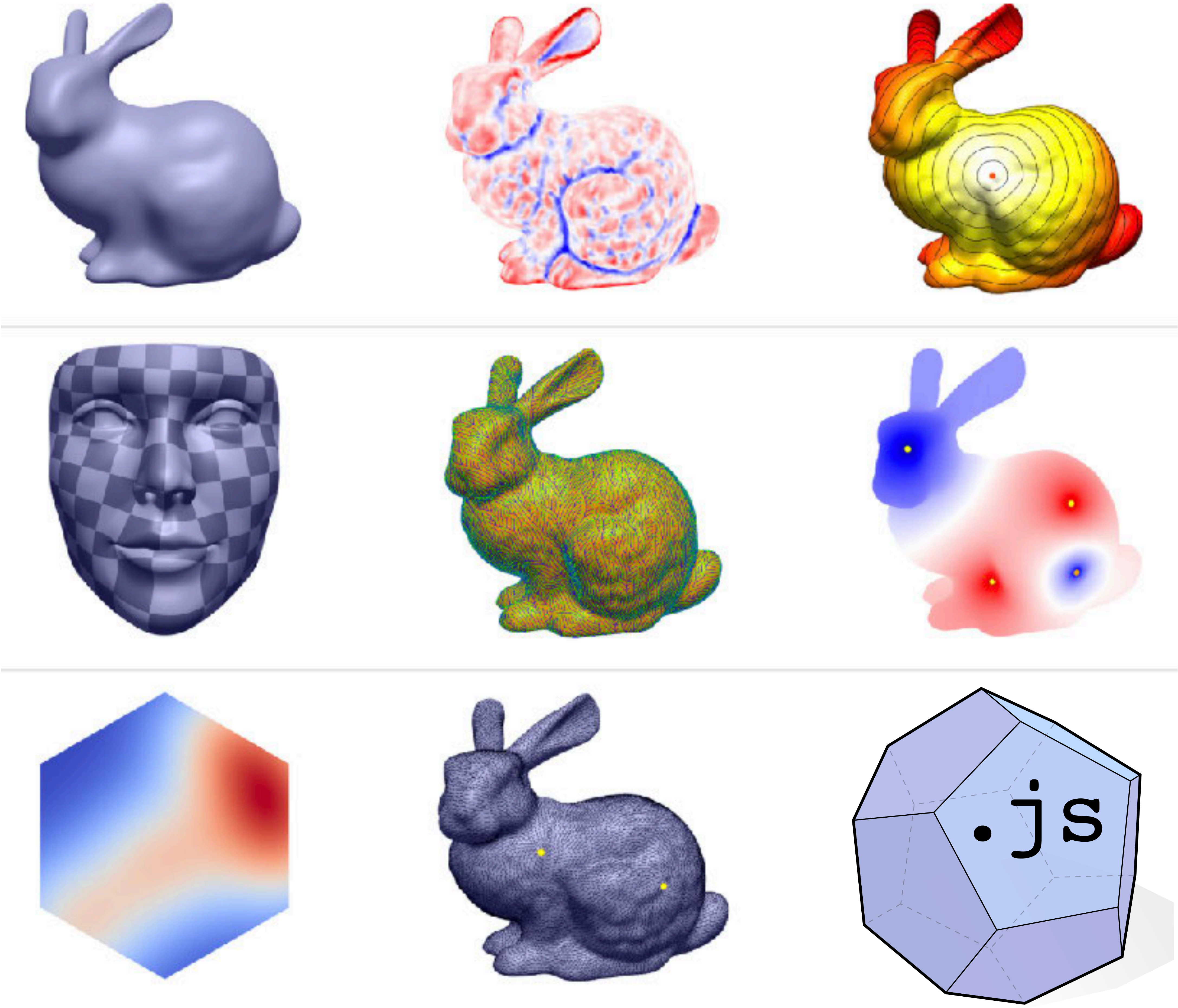


AMS Short Course
Discrete Differential Geometry

Joint Mathematics Meeting • San Diego, CA • January 2018

Demo Session — Overview

- **Goal:** give participants hands-on experience w/ DDG algorithms
- Implement (in web-based framework):
  - *discrete curvature*
  - *discrete Laplace-Beltrami*
- Experiment:
  - geodesic distance
  - direction fields
  - conformal mapping
  - …



Code Framework

- Open source web-based mesh processing framework
  - Write/debug algorithms in-browser (JavaScript)
  - Fast numerical libraries ported from C++
- **Why?**
  - Easy for students to access
  - Easy to share results online
  - Pretty good performance (within ~2–3x C++)
- Downside: language can be a bit "ugly" (*e.g.*, no operator overloading)

.js

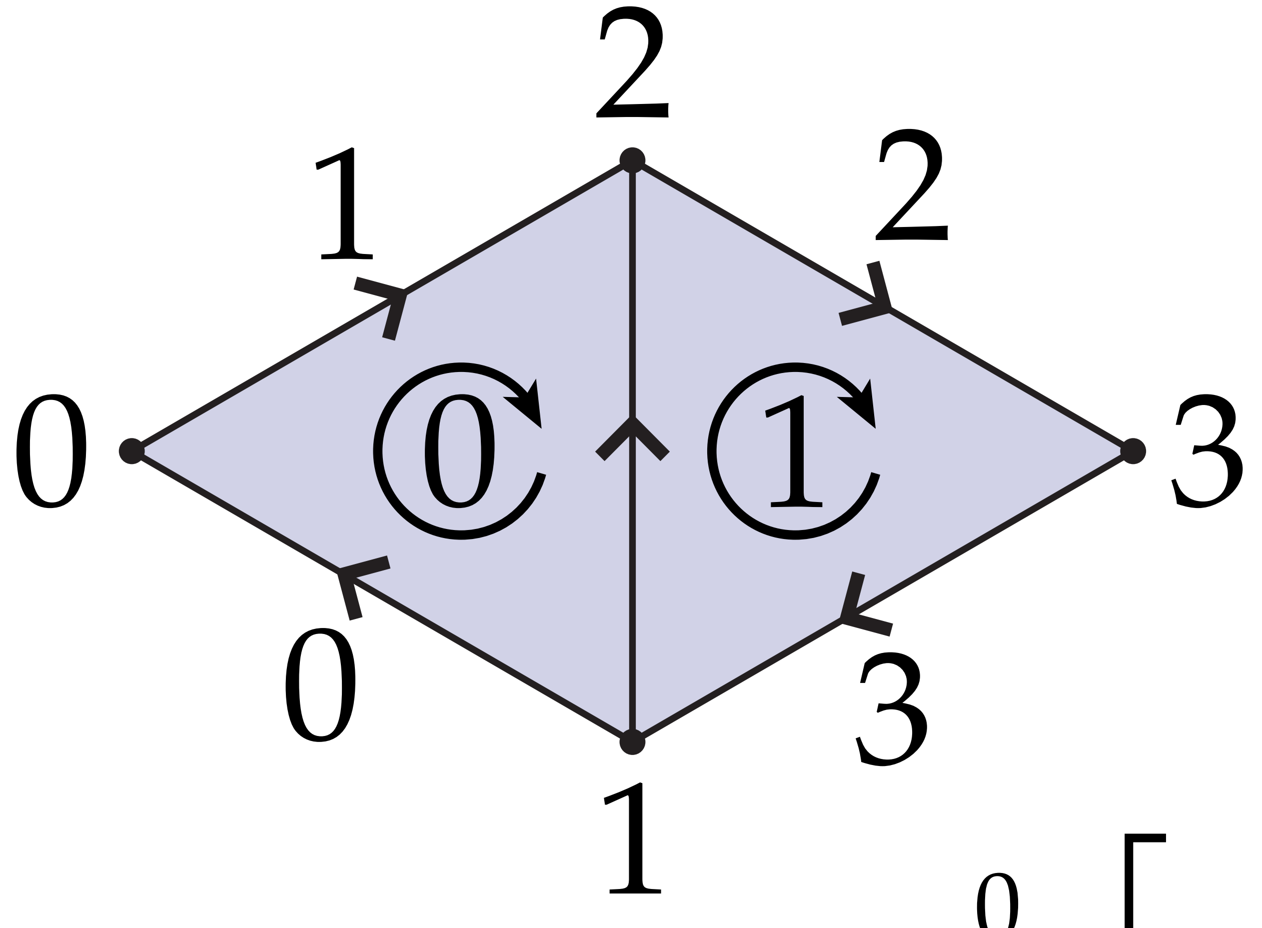
GEOMETRY-PROCESSING-JS

`http://geometry.cs.cmu.edu/js`

geometry-processing-js

Two main components (not much more needed for DDG algorithms!):

1. Mesh data structures
   - *halfedge mesh*
   - basic file I/O
2. Numerical linear algebra
   - *sparse & dense matrices*
   - fast linear solvers (`Eigen/emscripten`)

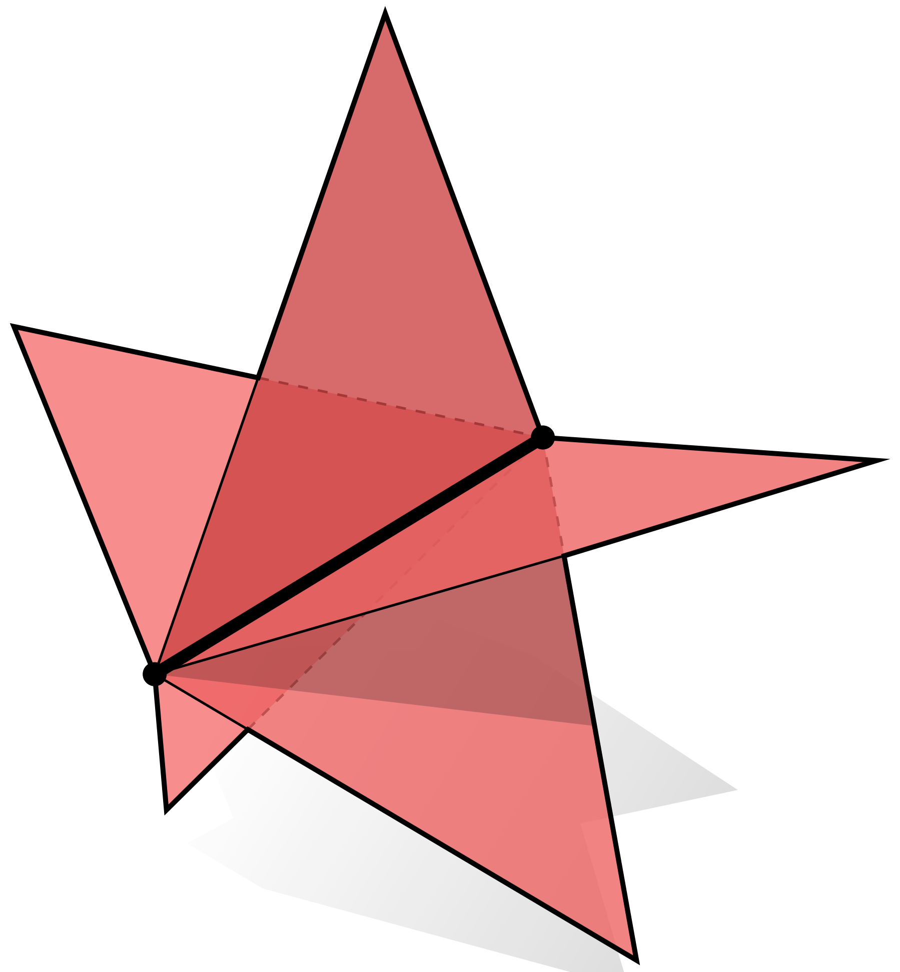


$$E^0 = \begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 1 & 0 \end{bmatrix} \end{array}$$

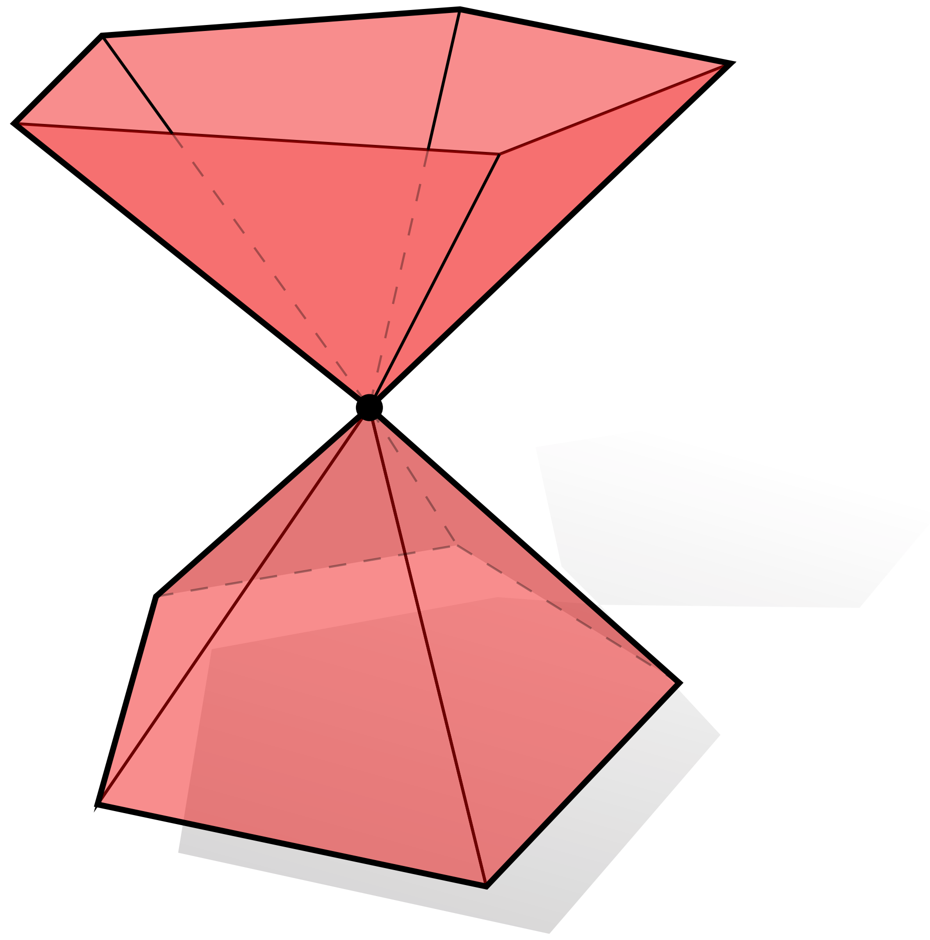
(Several example applications use WebGL for visualization)

Simplifying Assumption: Manifold Triangle Mesh

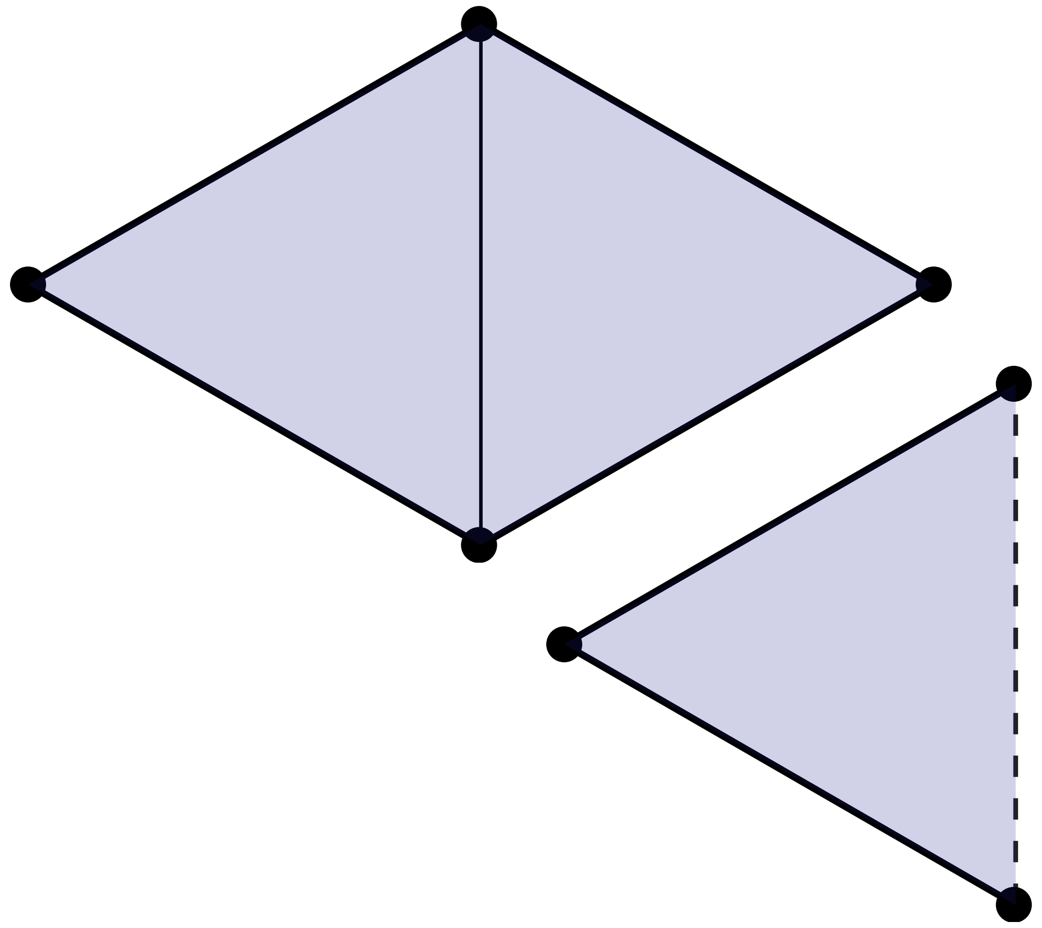
A simplicial 2-complex is **manifold** (with boundary) if every edge is contained in two (or one) triangles, and every vertex is contained in exactly one edge-connected cycle (or path) of triangles.



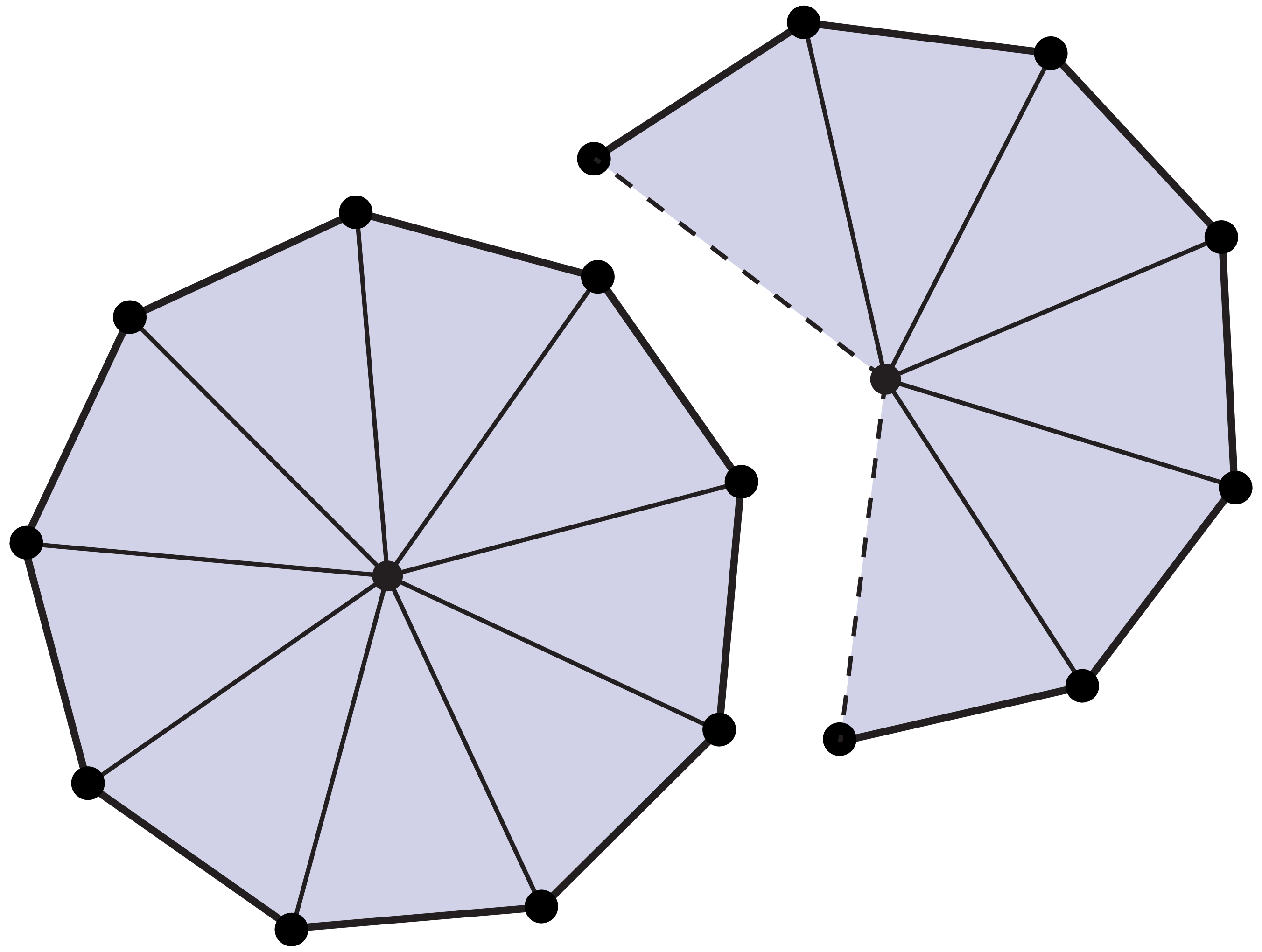
nonmanifold edge



nonmanifold vertex



nonmanifold edge (boundary edge)

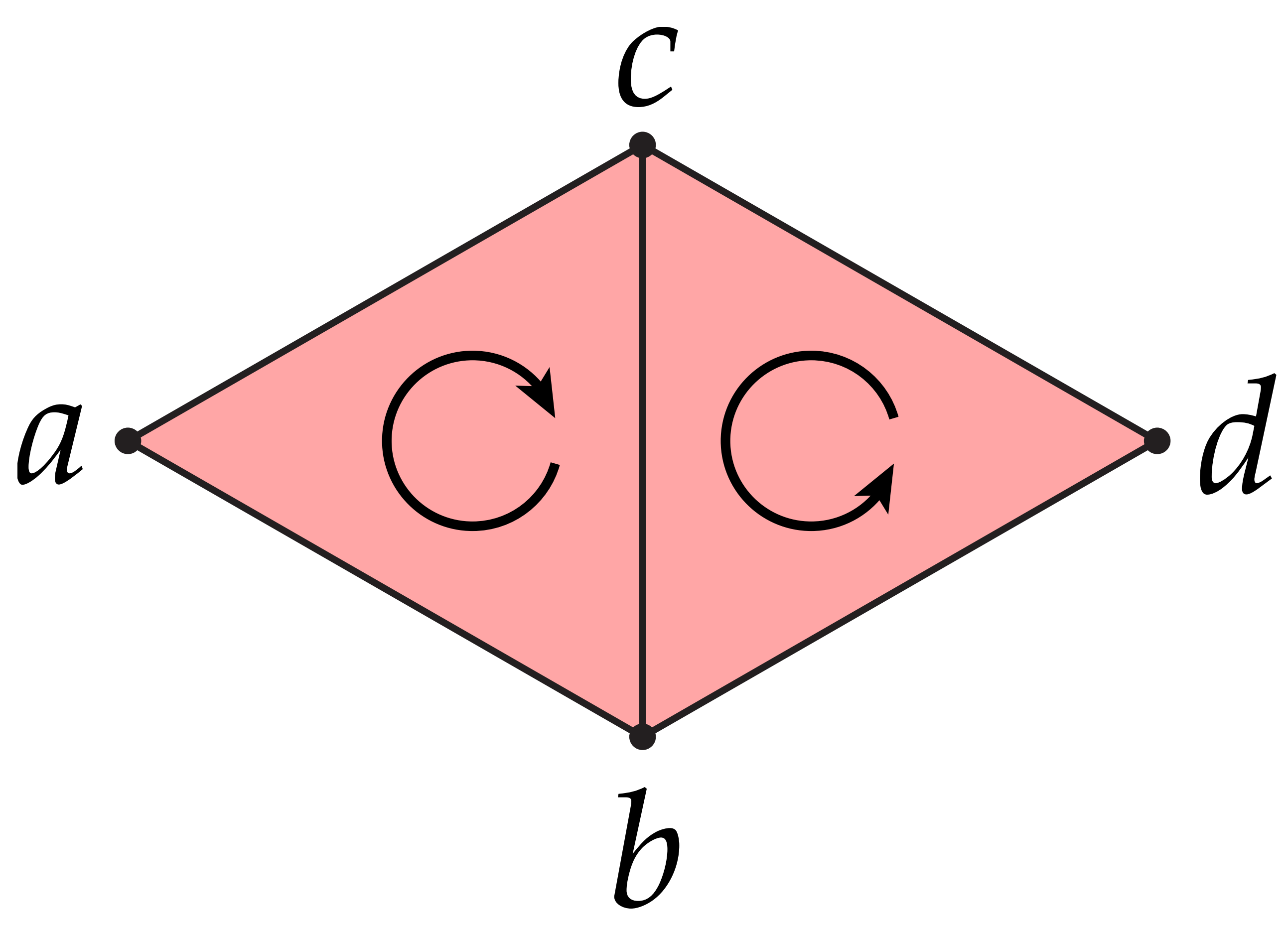


nonmanifold vertex (boundary vertex)

Simplifying Assumption: Oriented Triangle Mesh

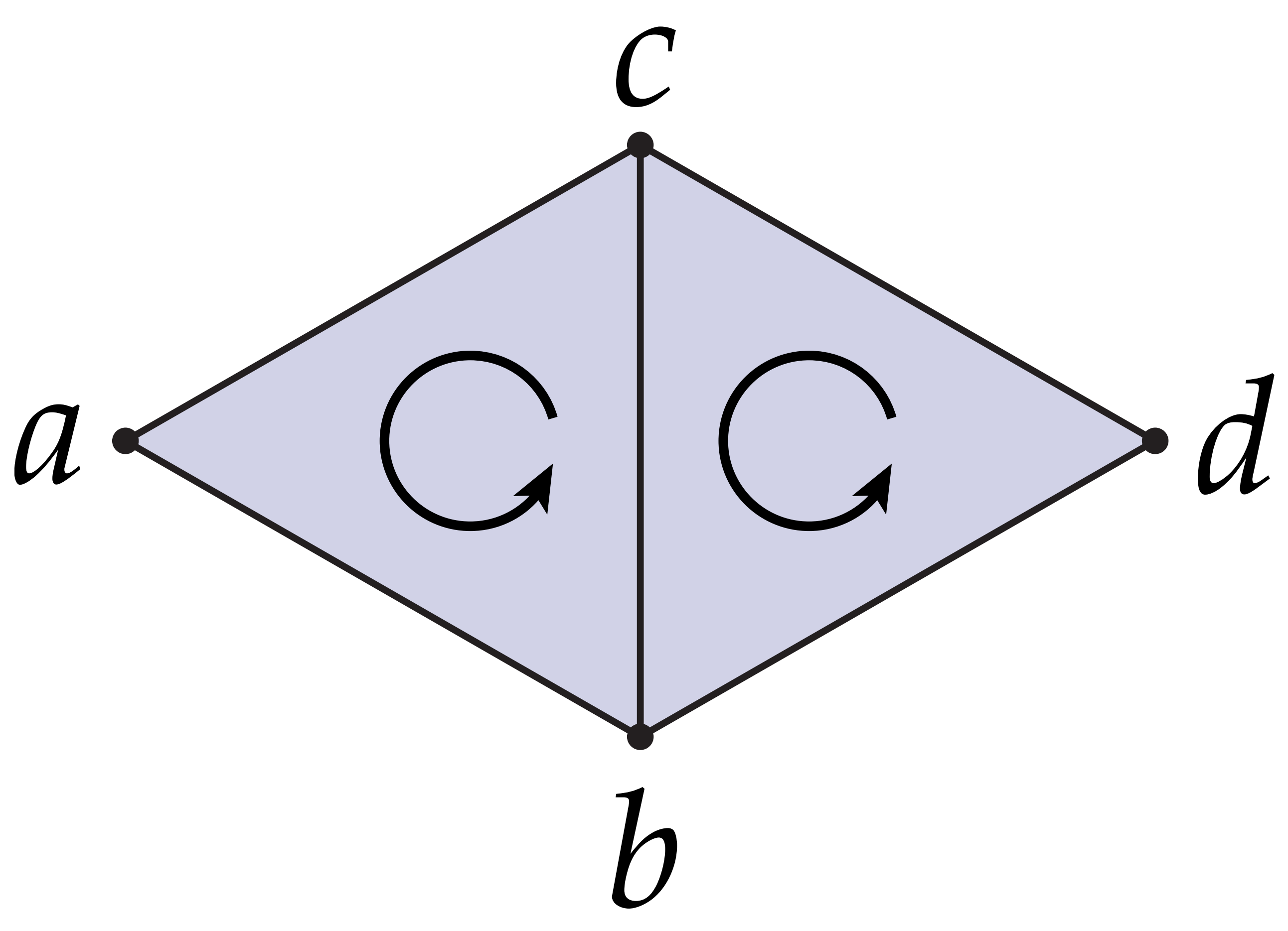
An **orientation** of a simplex is an ordering of its vertices, up to even permutation. A manifold triangle mesh is **orientable** if its triangles can be given a globally consistent ordering.

$(a, c, b), (c, b, d)$



not consistent

$(a, b, c), (c, b, d)$



consistent

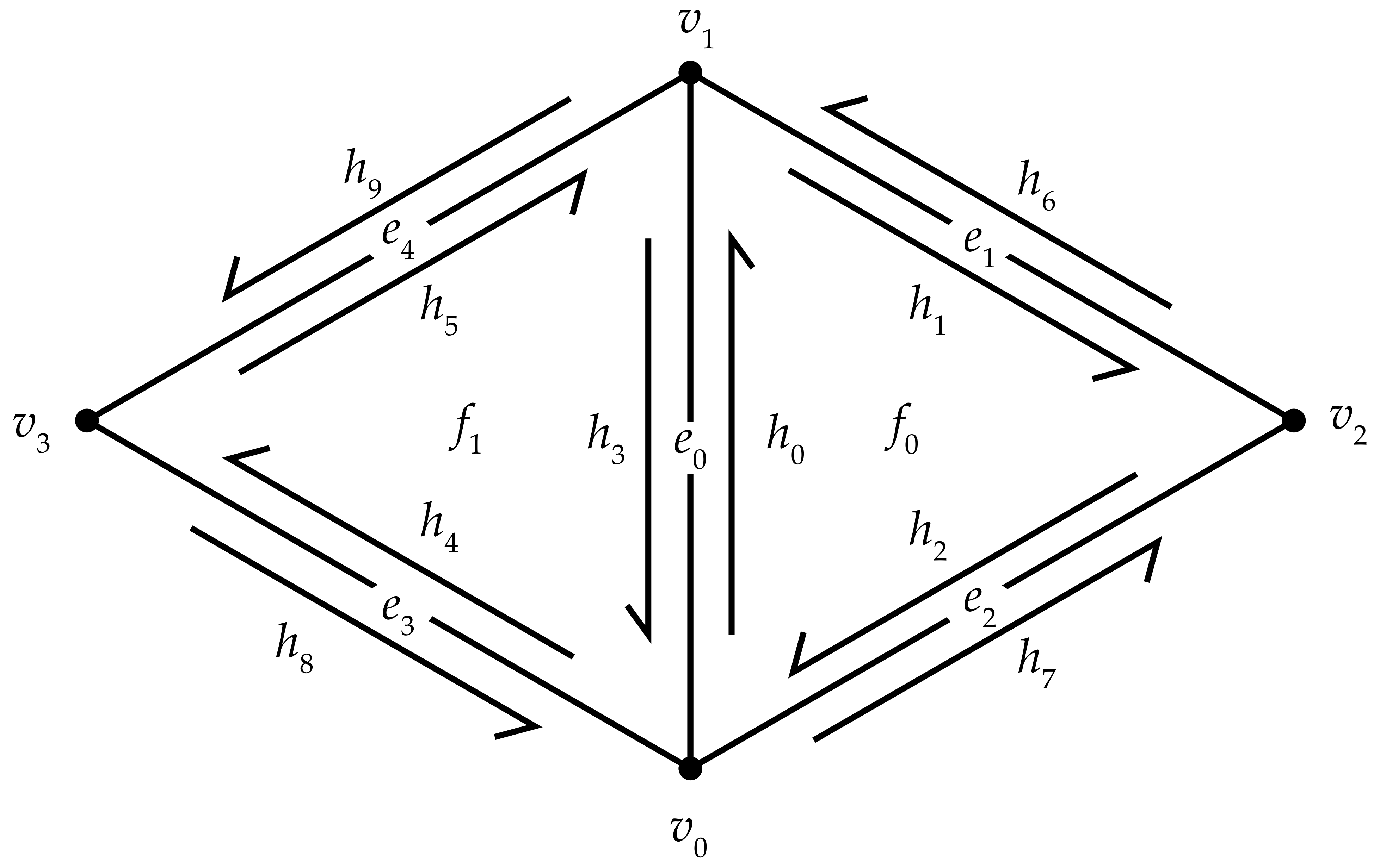
Halfedge Mesh

Definition. Let H be any set with an even number of elements, let $\rho : H \to H$ be any permutation of H, and let $\eta : H \to H$ be a fixed point free involution, *i.e.*, $\eta \circ \eta = \text{id}$ and $\eta(h) \neq h$ for any $h \in H$. Then (H, ρ, η) is a *halfedge mesh*, the elements of H are called *halfedges*, the orbits of ρ are *faces*, the orbits of η are *edges*, and the orbits of $\rho \circ \eta$ are *vertices*.

Fact. Every halfedge mesh describes a compact oriented topological surface.

$$\underset{\textbf{“next”}}{(h_0, \ldots, h_9) \overset{\rho}{\mapsto}} (h_1, h_2, h_0, h_4, h_5, h_3, h_9, h_6, h_7, h_8)$$

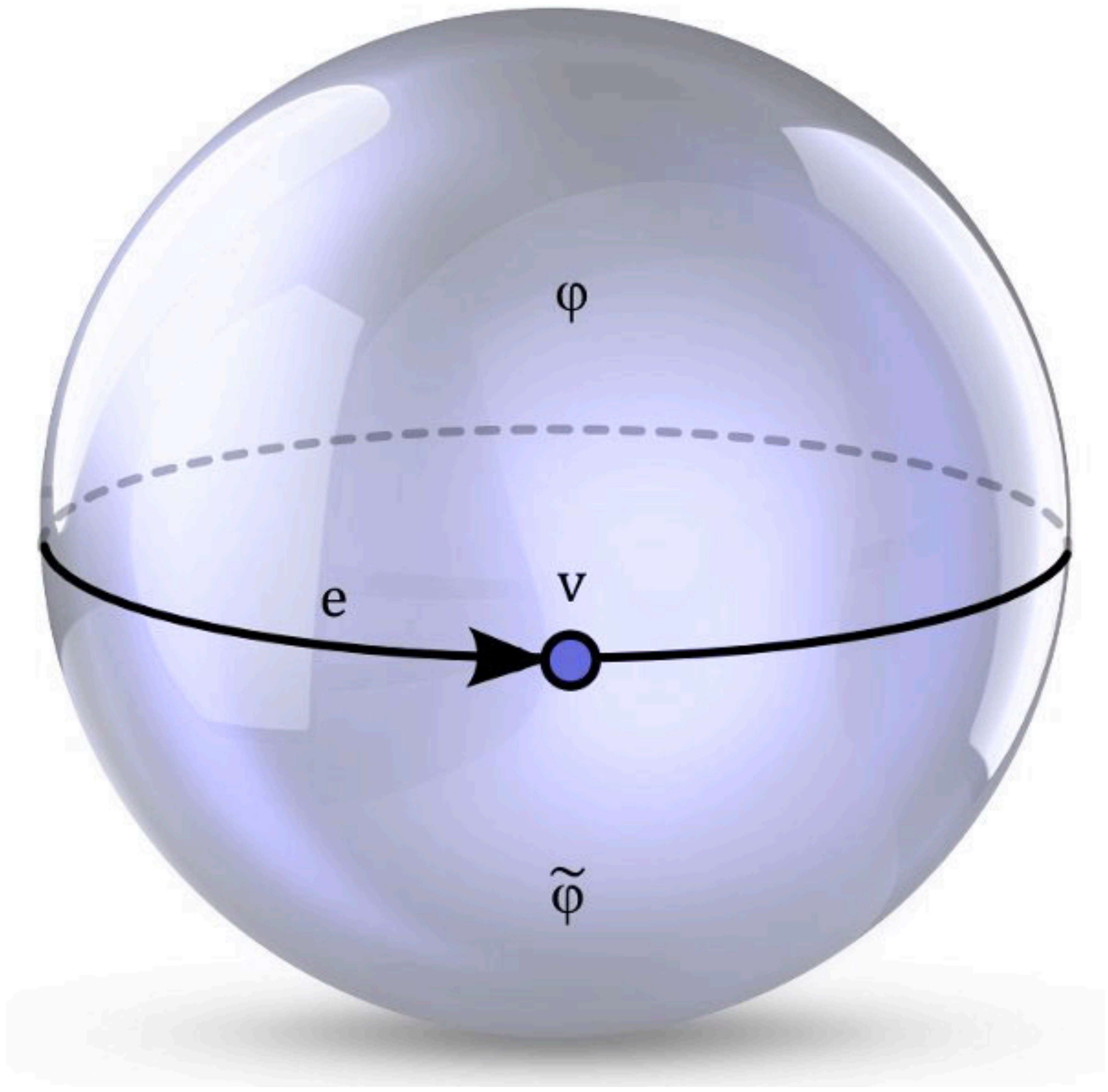
$$\underset{\textbf{“twin”}}{(h_0, \ldots, h_9) \overset{\eta}{\mapsto}} (h_3, h_6, h_7, h_0, h_8, h_9, h_1, h_2, h_4, h_5)$$



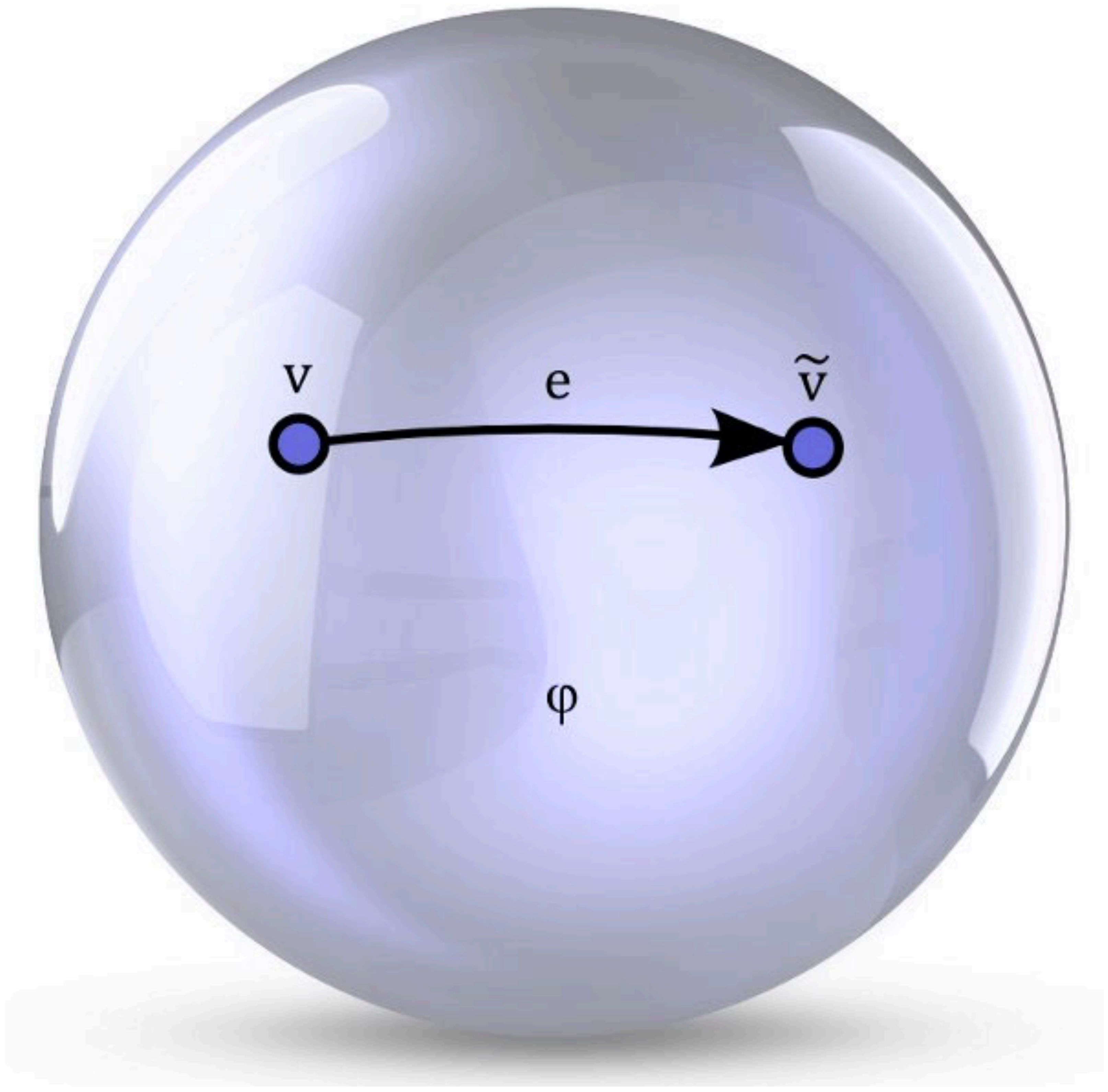
Halfedge Mesh—Example

Smallest examples (two half edges):

(images courtesy U. Pinkall)



$$\rho(1,2) = (1,2)$$
$$\eta(1,2) = (2,1)$$



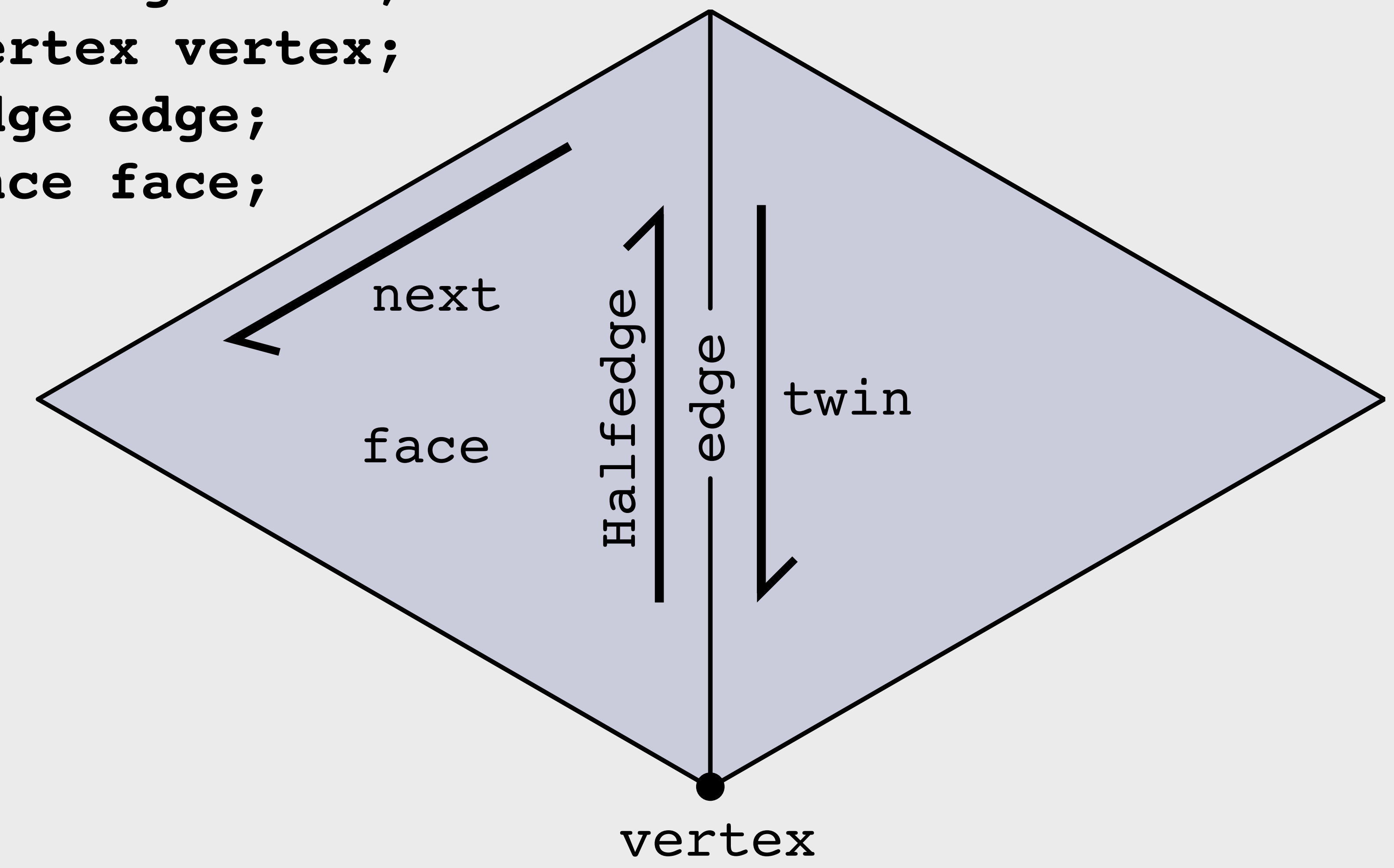
$$\rho(1,2) = (2,1)$$
$$\eta(1,2) = (2,1)$$

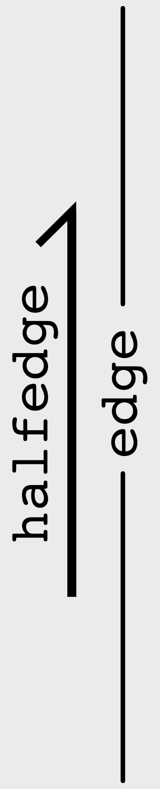
Practical Halfedge Data Structure

Basic idea: each edge gets split into two *half edges.*

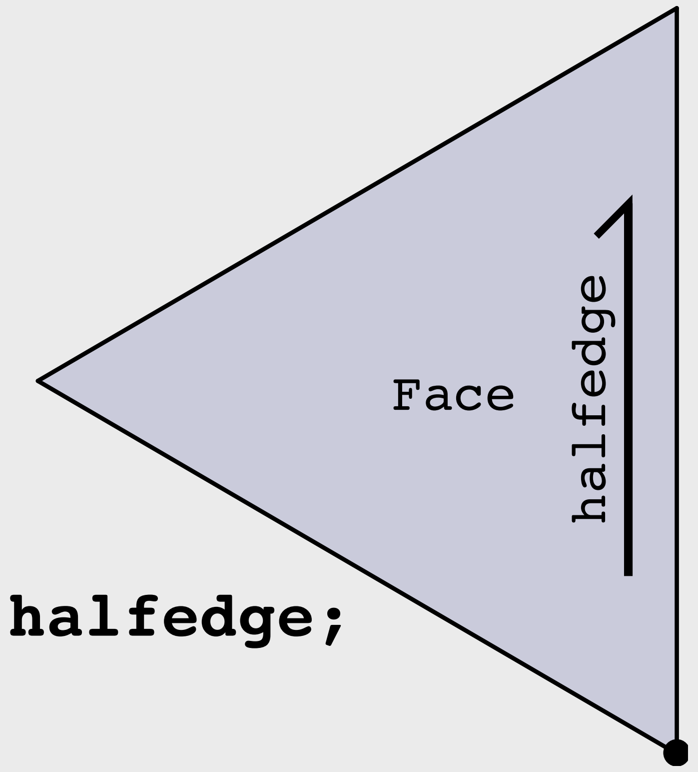
- Half edges act as “glue” between mesh elements.
- All other elements know only about a single half edge.

```
class Halfedge
{
   Halfedge twin;
   Halfedge next;
   Vertex vertex;
   Edge edge;
   Face face;
};
```

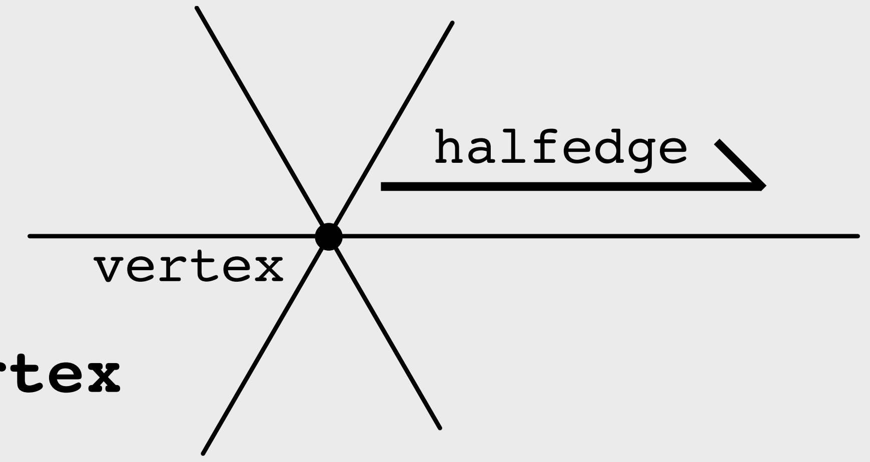





```
class Edge
{
   Halfedge halfedge;
};
```




```
class Face
{
   Halfedge halfedge;
};
```

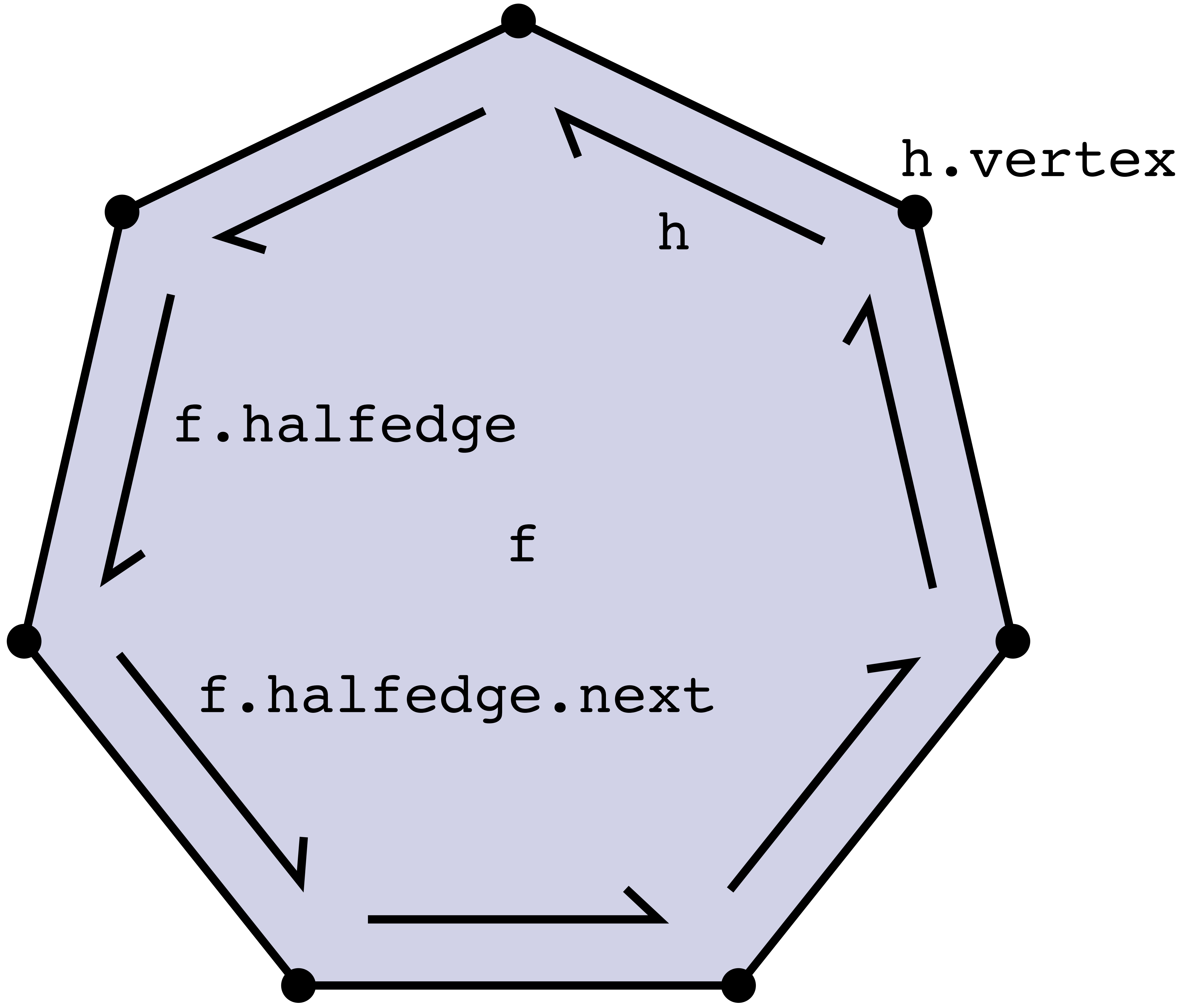



```
class Vertex
{
   Halfedge halfedge;
};
```

Traversing a Halfedge Mesh

- Key feature of halfedge mesh: easy traversal of nearby elements
- *E.g.*, suppose we want to visit all vertices of a face:

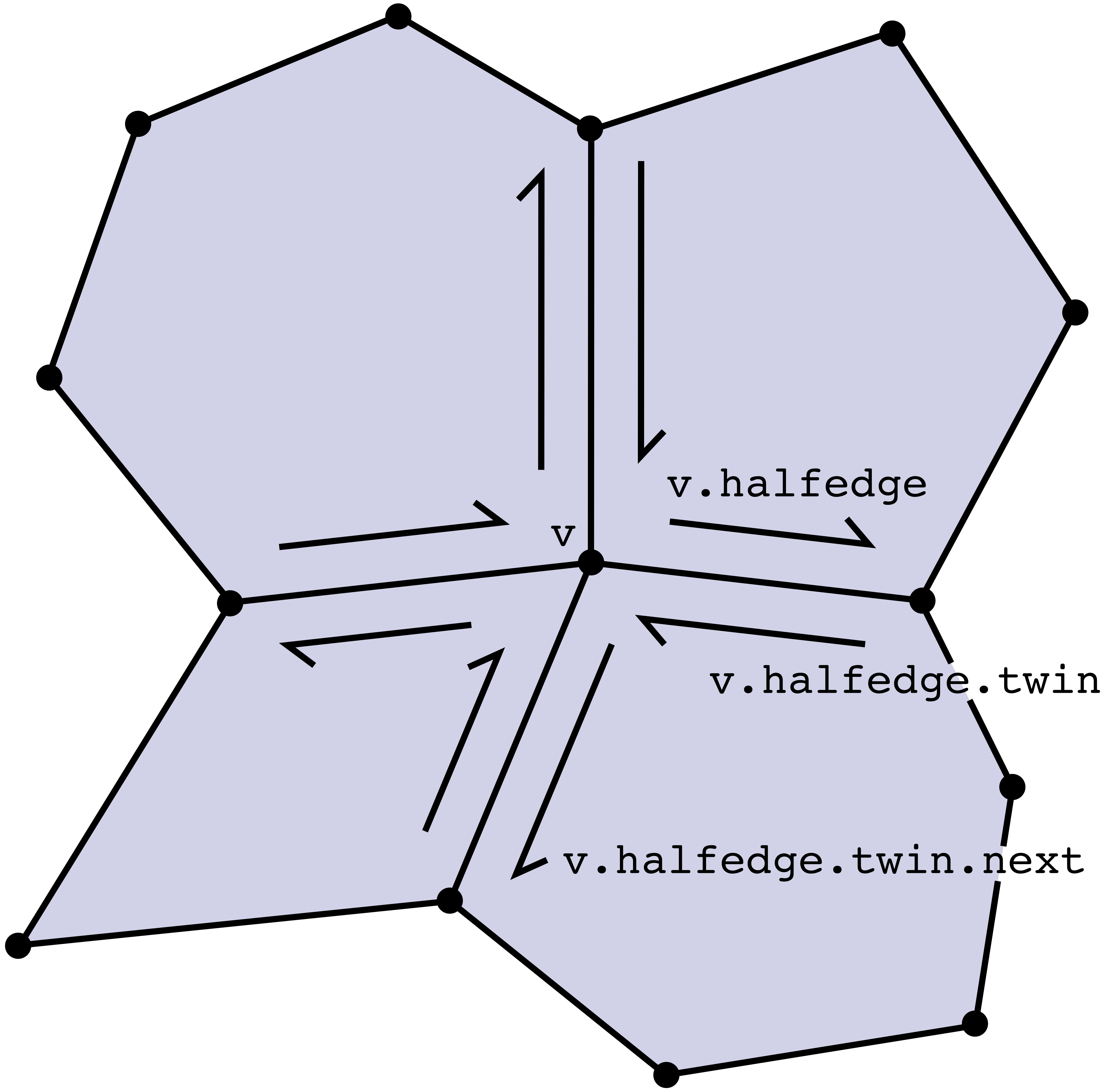
```
Face f;

let h = f.halfedge;
do
{
   let u = h.vertex;
   // (do something with u)
   h = h.next;
}
while( h != f.halfedge );
```



Traversing a Halfedge Mesh

- Similarly, suppose we want to visit all vertices adjacent to a given vertex:

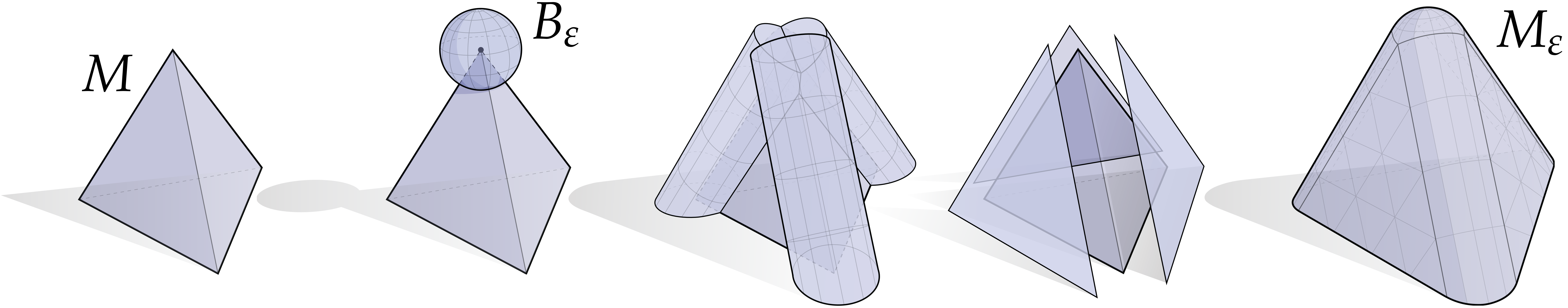
```
Vertex v;

let h = v.halfedge;
do
{
   u = h.twin.vertex;
   // (do something with u)
   h = h.twin.next;
}
while( h != v.halfedge );
```



Activity 1: Discrete Curvature

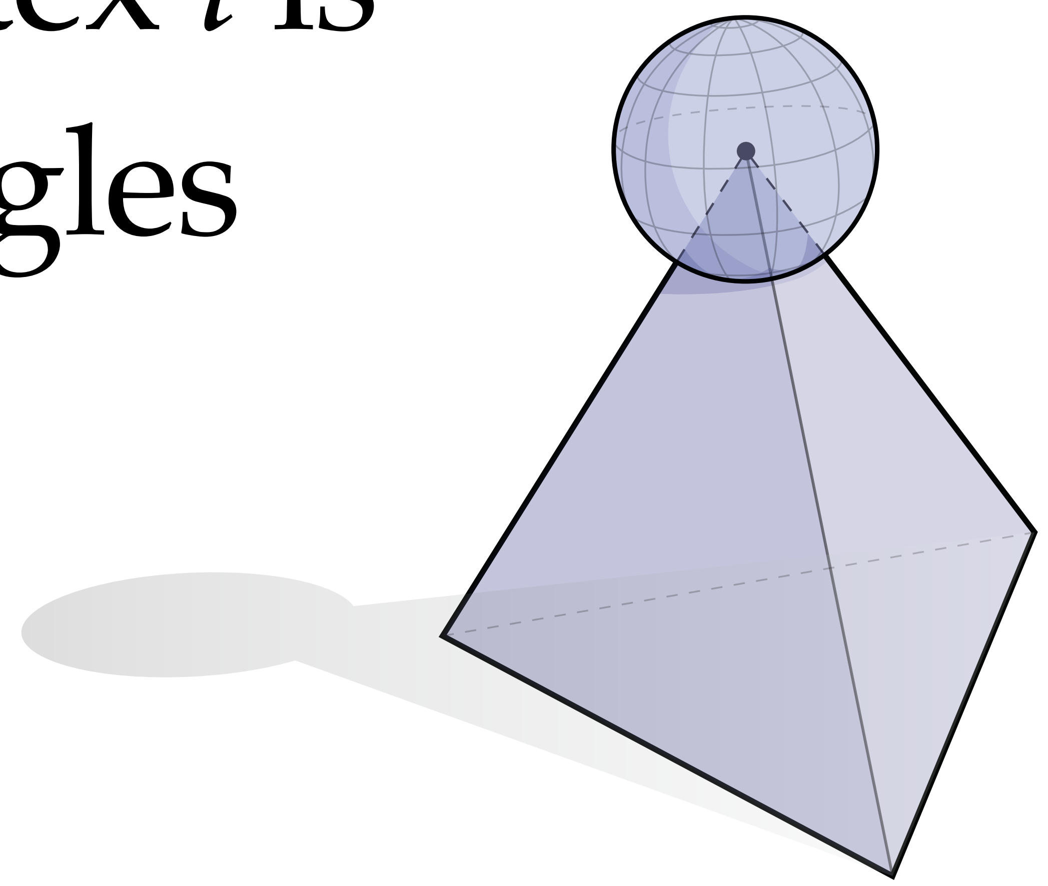
Curvature of a Simplicial Surface

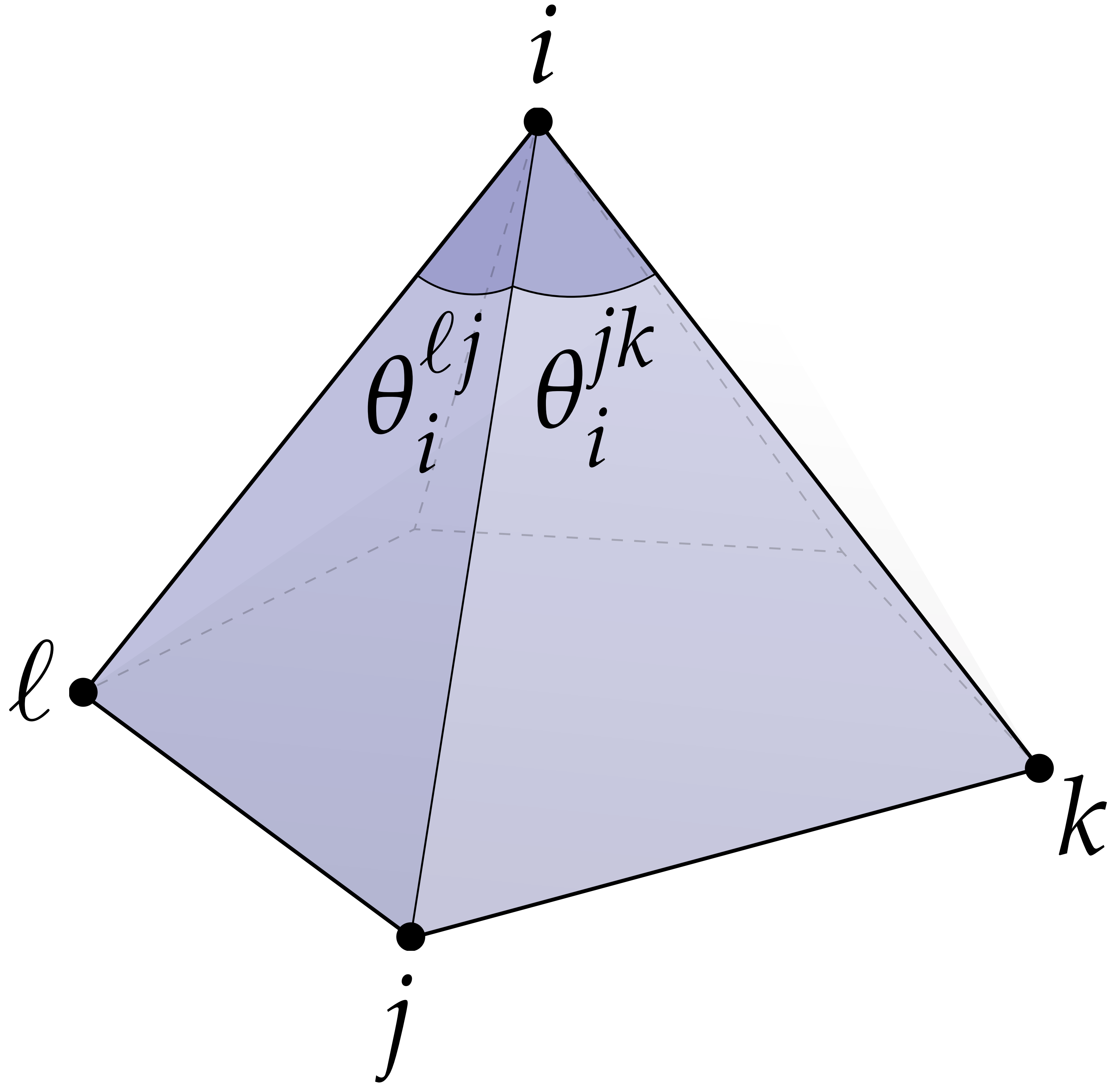
- How can we define the curvature of a simplicial surface M?
- One possibility (of many): use curvature of *“mollified”* surface M_ε
  - take Minkowski sum with ball B_ε of radius ε
  - derive expression for curvatures (*Steiner formula*)
  - take limit as ε goes to zero



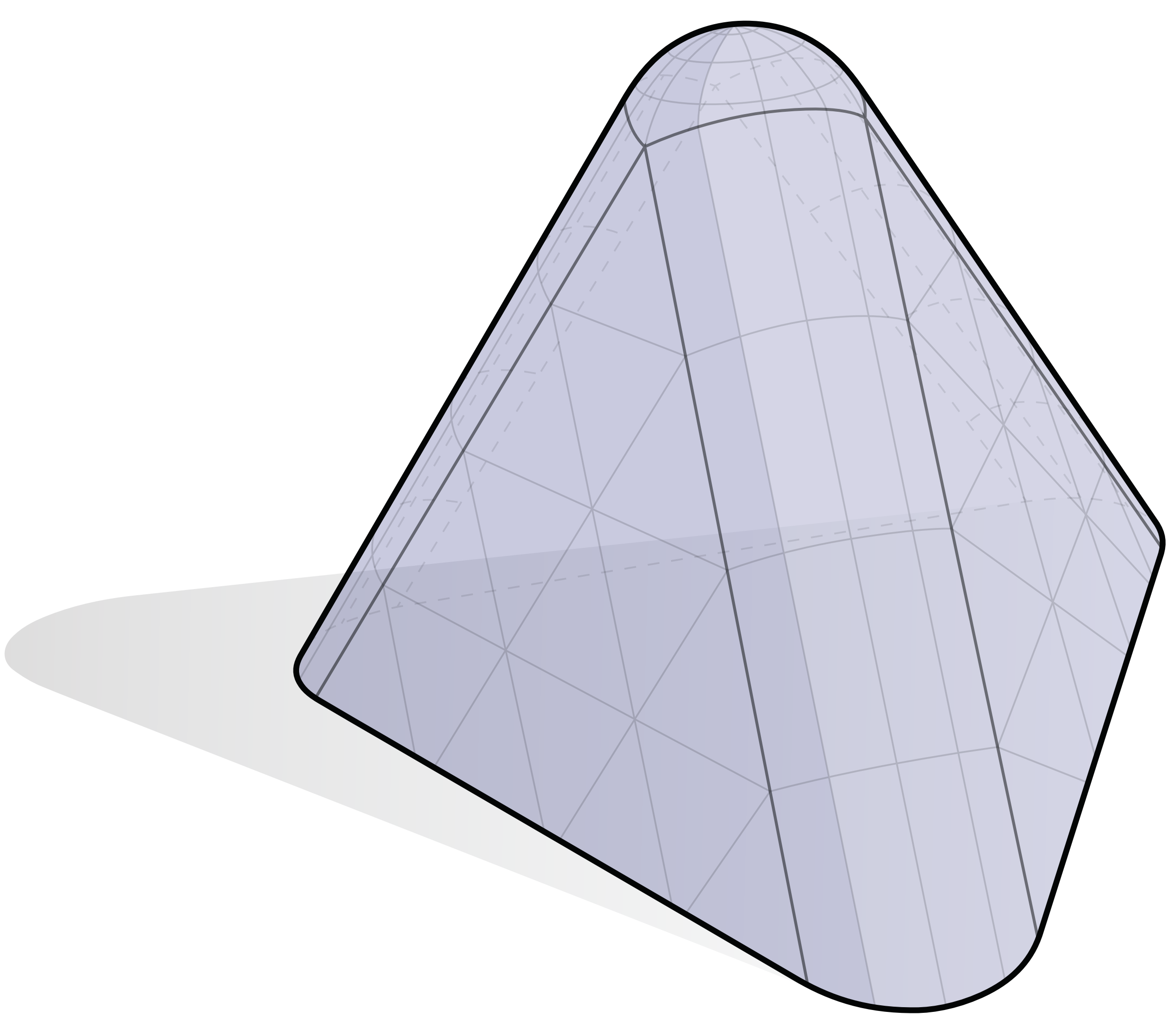
Discrete Gaussian Curvature

Total Gaussian curvature of region associated with a vertex i is equal to the **angle defect**, *i.e.*, the deviation of interior angles around the vertex from the Euclidean angle sum 2π:





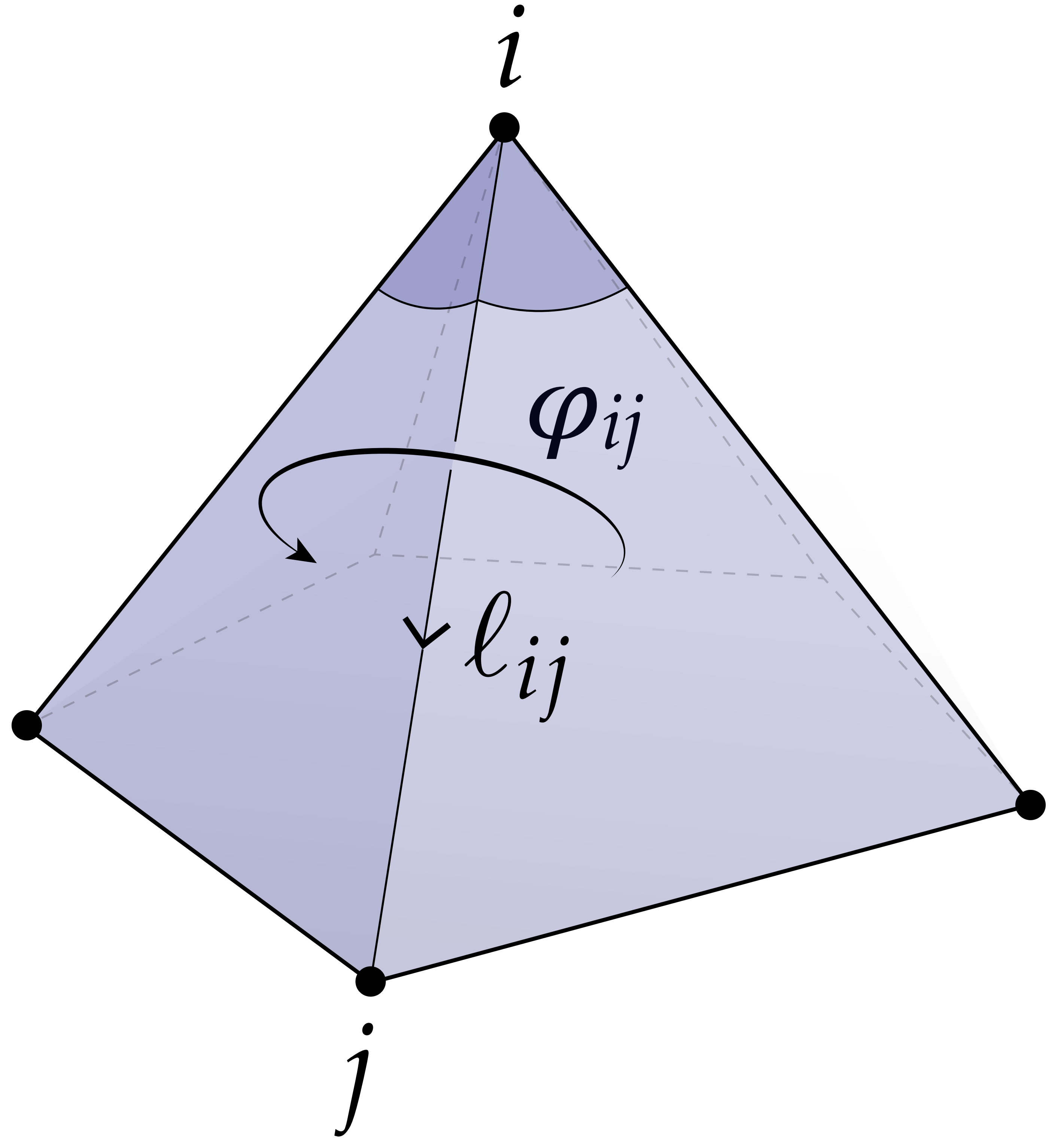
$$K_i := 2\pi - \sum_{ipq} \theta_i^{pq}$$



(**Intuition:** how "flat" is the vertex?)

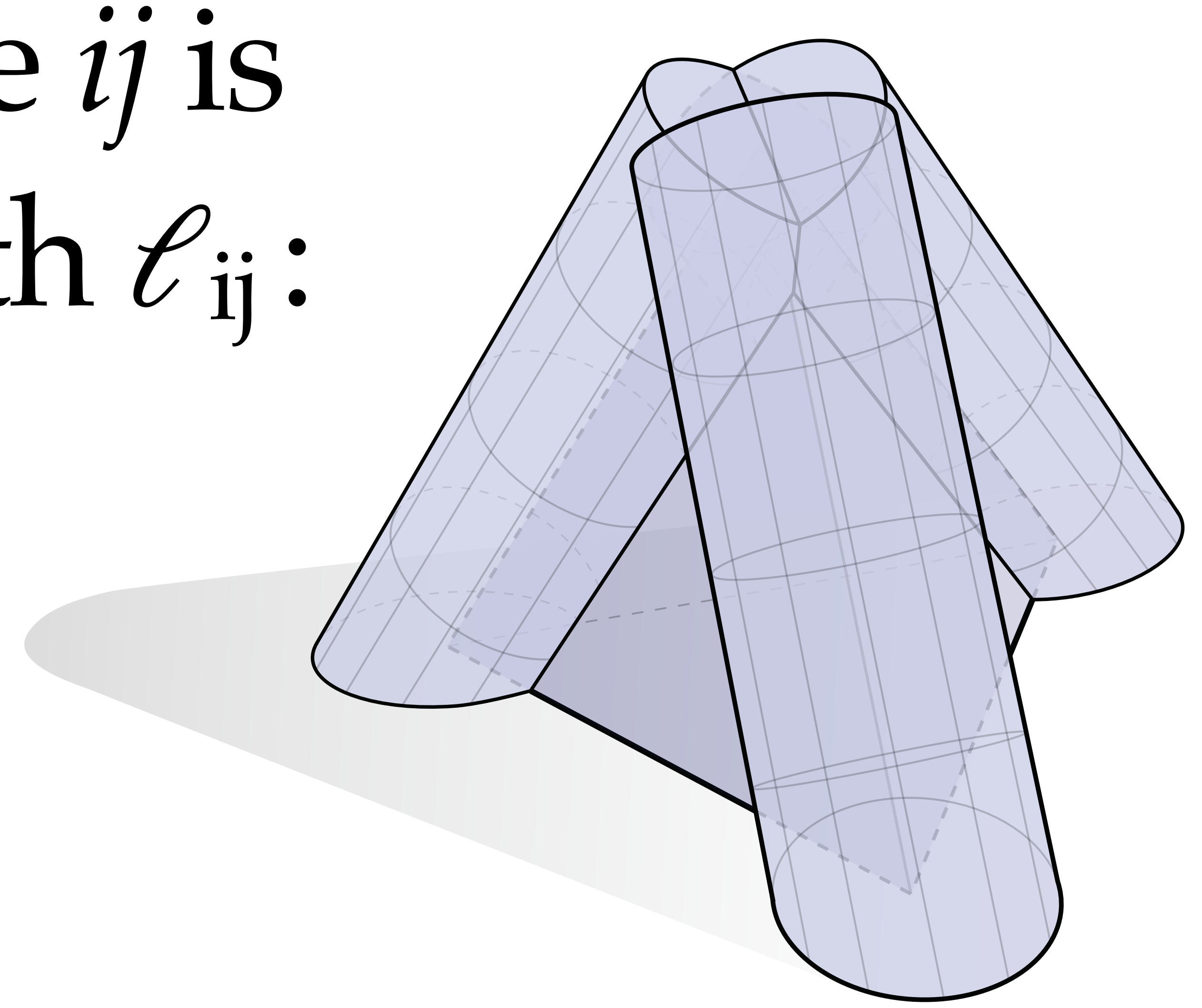
Discrete Mean Curvature

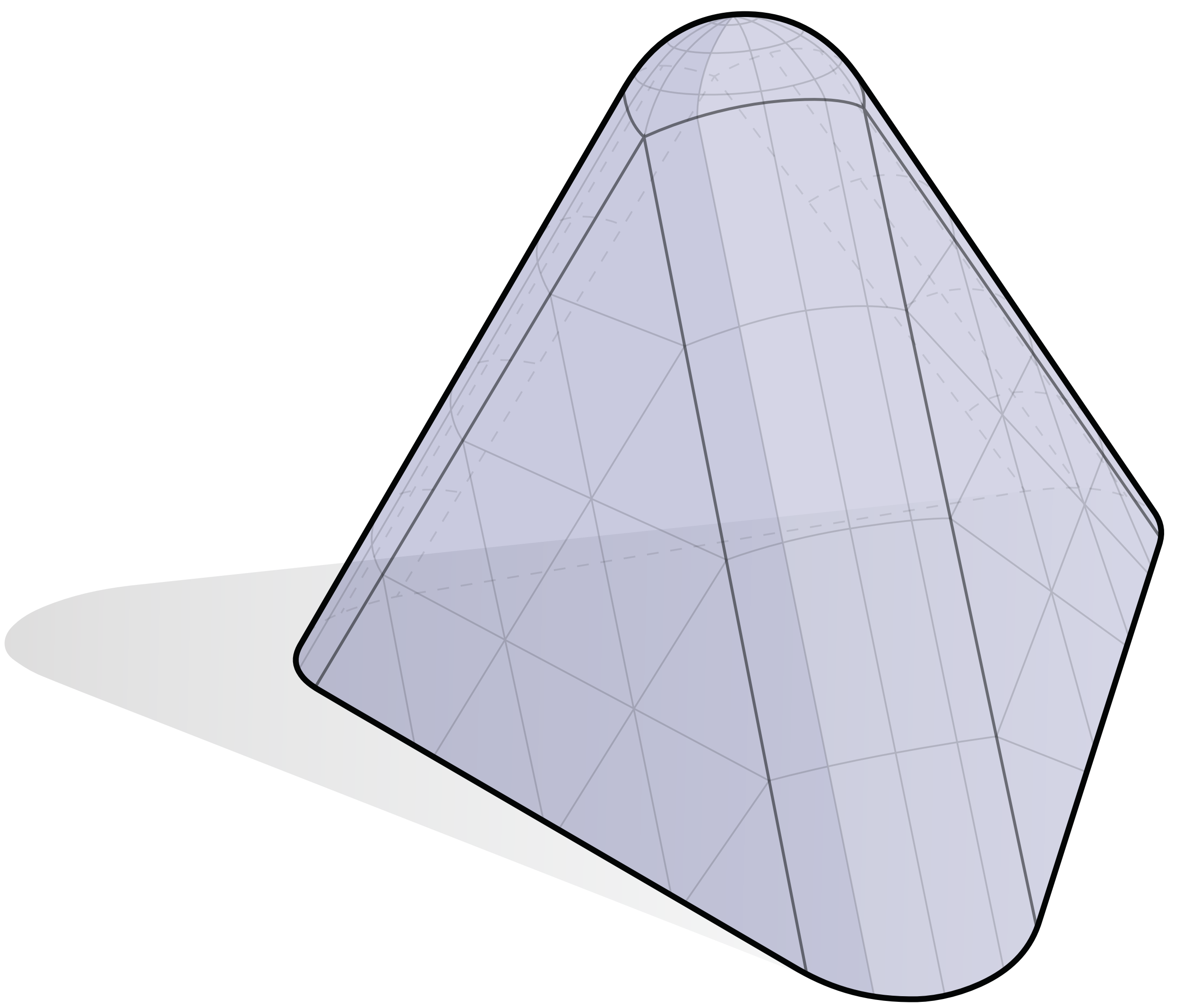
Total mean curvature of region associated with an edge ij is equal to half the dihedral angle φ_{ij} times the edge length ℓ_{ij}:



$$H_{ij} := \frac{1}{2}\varphi_{ij}\ell_{ij}$$

$$\Rightarrow H_i := \frac{1}{4}\sum_{ij}\varphi_{ij}\ell_{ij}$$

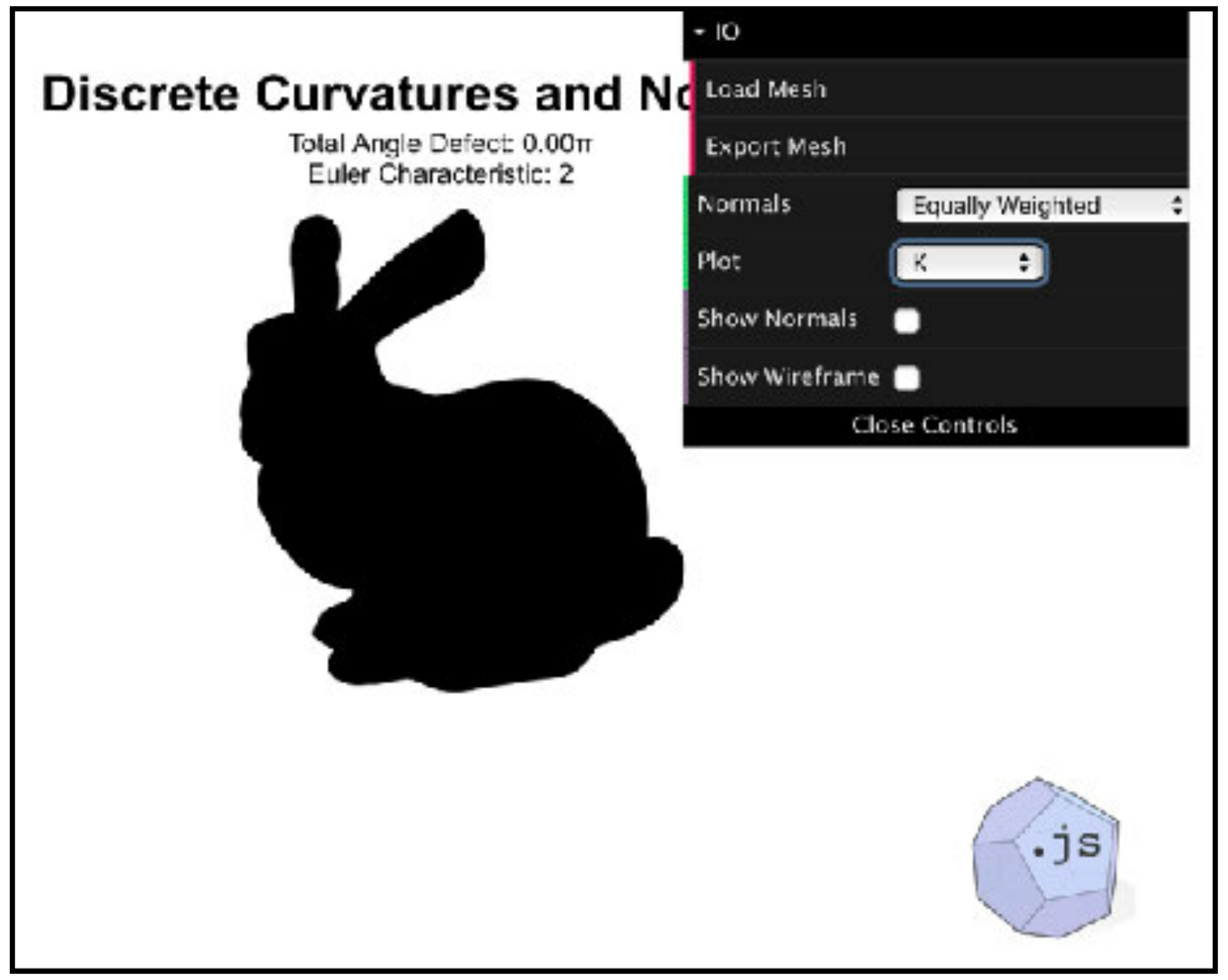




(**Intuition:** how “bent” is the edge?)

Implementing Discrete Curvature

- Two directories:
  - **`ddg-js_skeleton/`** — partial implementation (*we'll fill this one in*)
  - `ddg-js_solution/` — working implementation (*for reference*)
- Documentation in `docs/index.html`
- For now, open…
  - in **Chrome**: `projects/discrete-curvatures-and-normals/index.html`
  - in **text editor**: `core/geometry.js`
- In `geometry.js`, search for two methods:
  - `scalarGaussCurvature(v)`
  - `scalarMeanCurvature(v)`

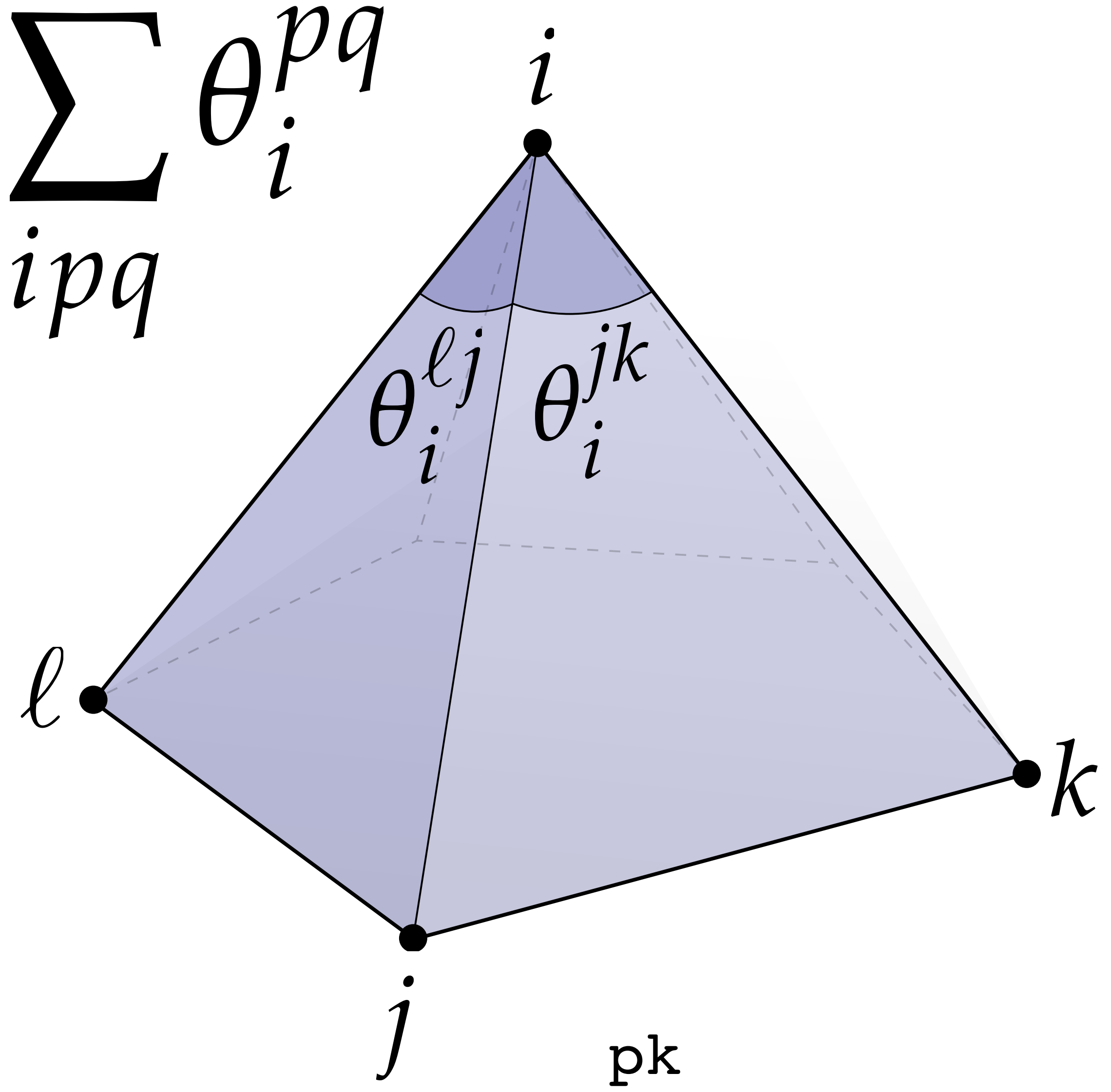


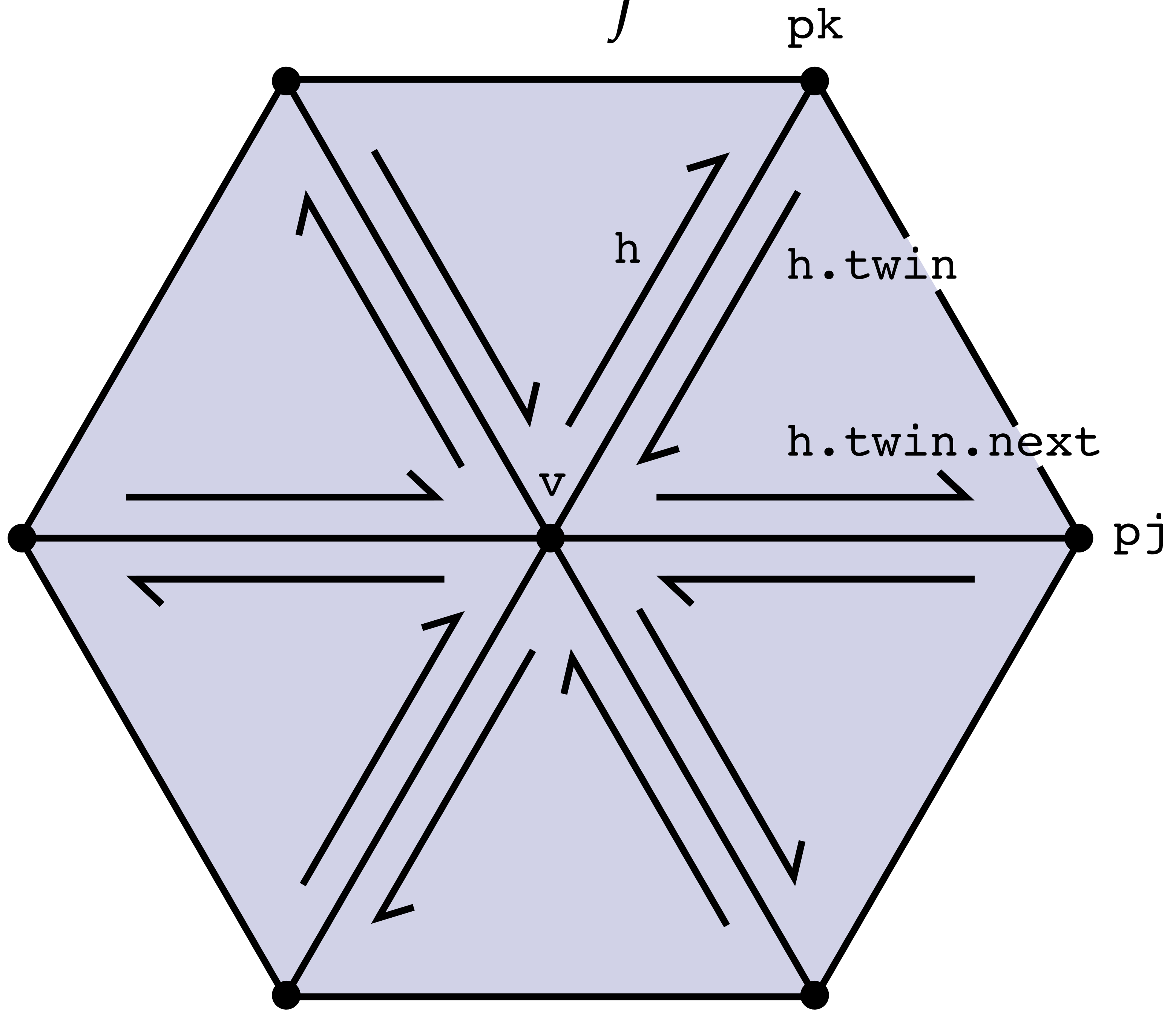
Implementing Discrete Gauss Curvature

One implementation of **`scalarGaussCurvature(v)`**:

$$K_i := 2\pi - \sum_{ipq} \theta_i^{pq}$$

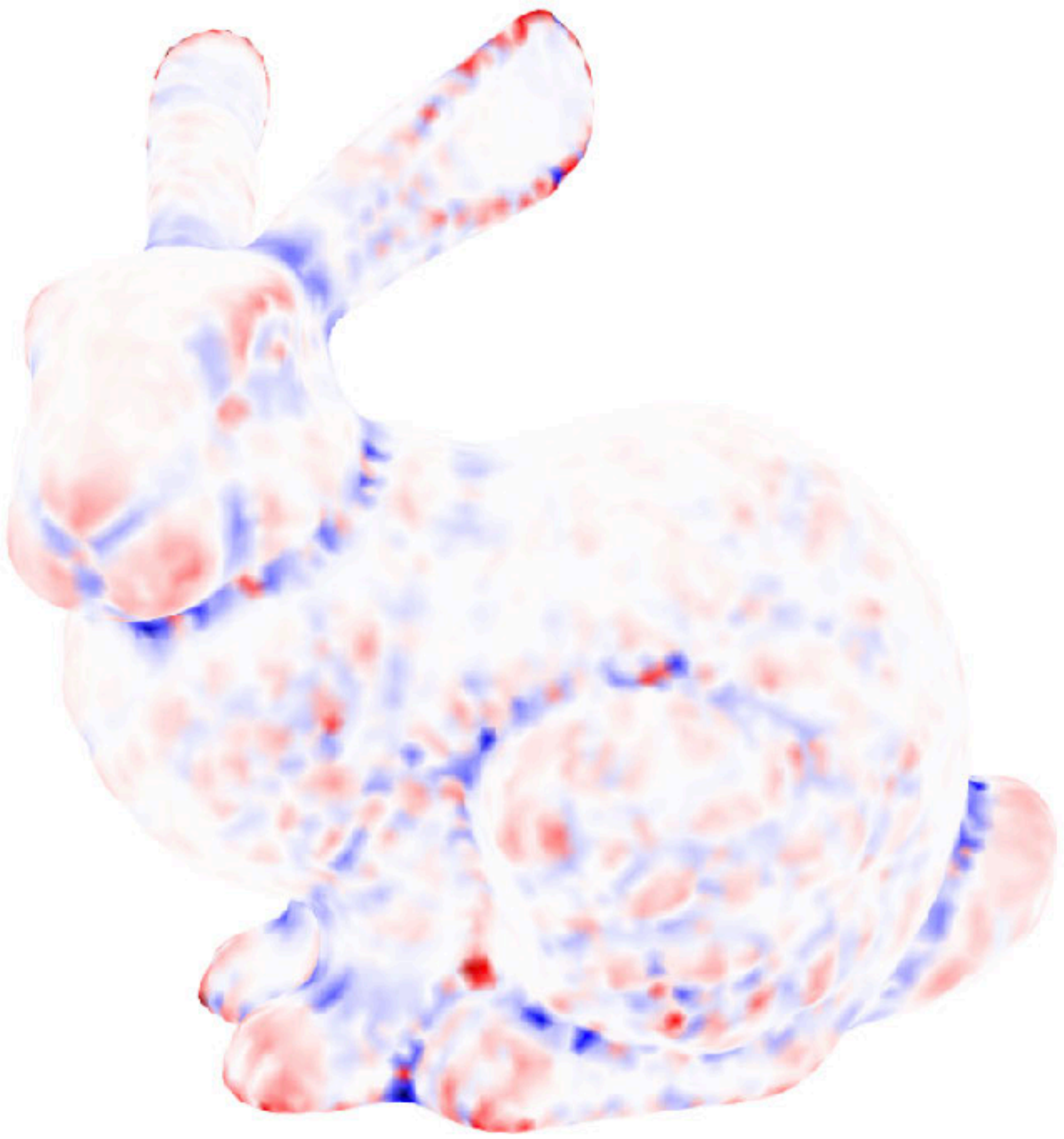
```
let angleSum = 0.0; // will accumulate sum of angles
let h = v.halfedge; // start with any halfedge
do
{
   // get vertex positions for current triangle ijk
   // ("this" refers to the currently loaded mesh)
   let pi = this.positions[h.vertex];
   let pj = this.positions[h.next.vertex];
   let pk = this.positions[h.next.next.vertex];

   // compute interior angle at vertex i
   let u = (pj.minus(pi)).unit(); // unit vector from pi to pj
   let v = (pk.minus(pi)).unit(); // unit vector from pi to pk
   let theta = Math.acos( u.dot(v) ); // angle between u and v

   angleSum += theta; // accumulate angle sum
   h = h.twin.next; // move to next halfedge
}
while( h != v.halfedge ); // stop when we get back to beginning

return 2.0*Math.PI - angleSum; // return defect
```

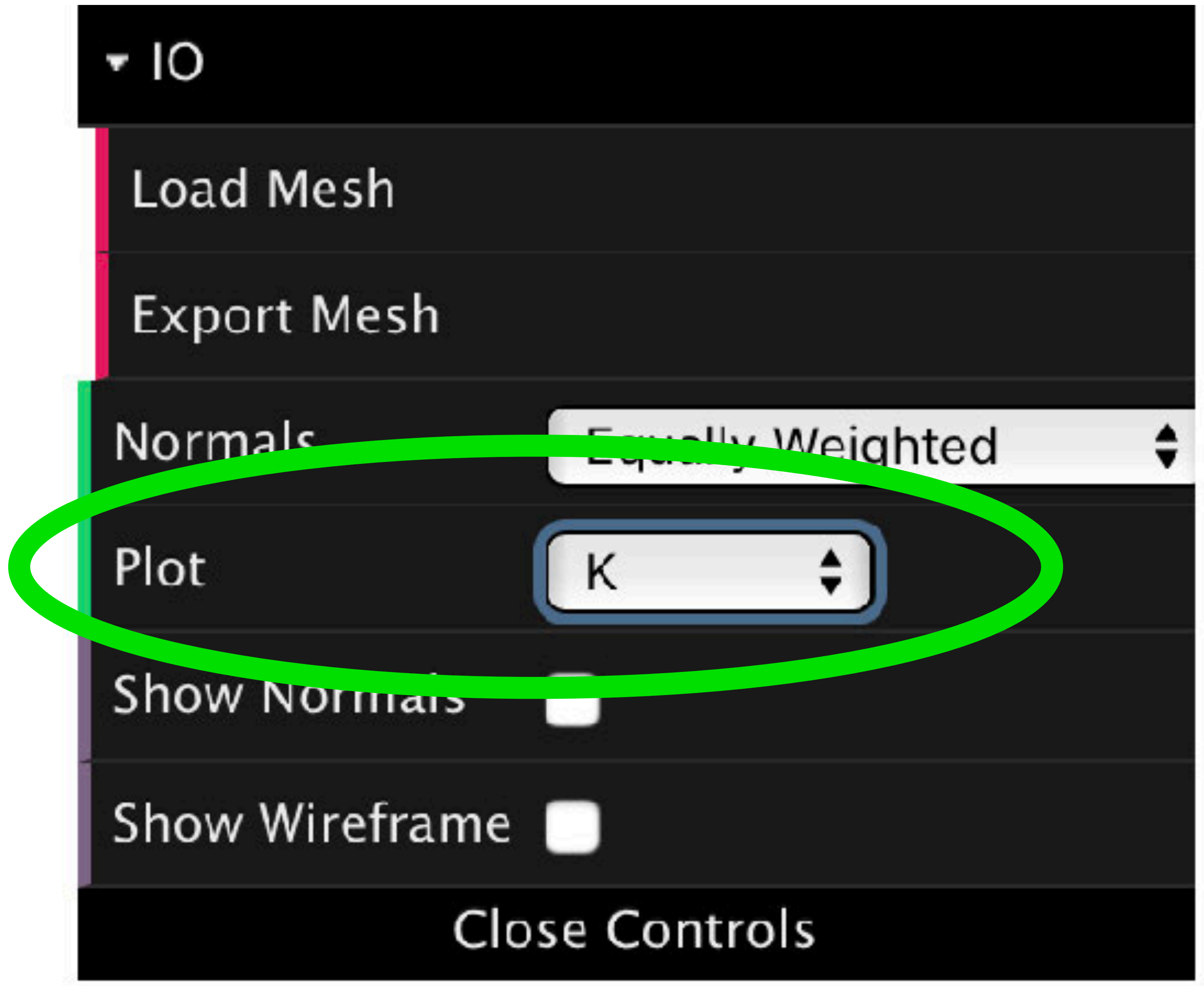




Discrete Gaussian Curvature — Visualized

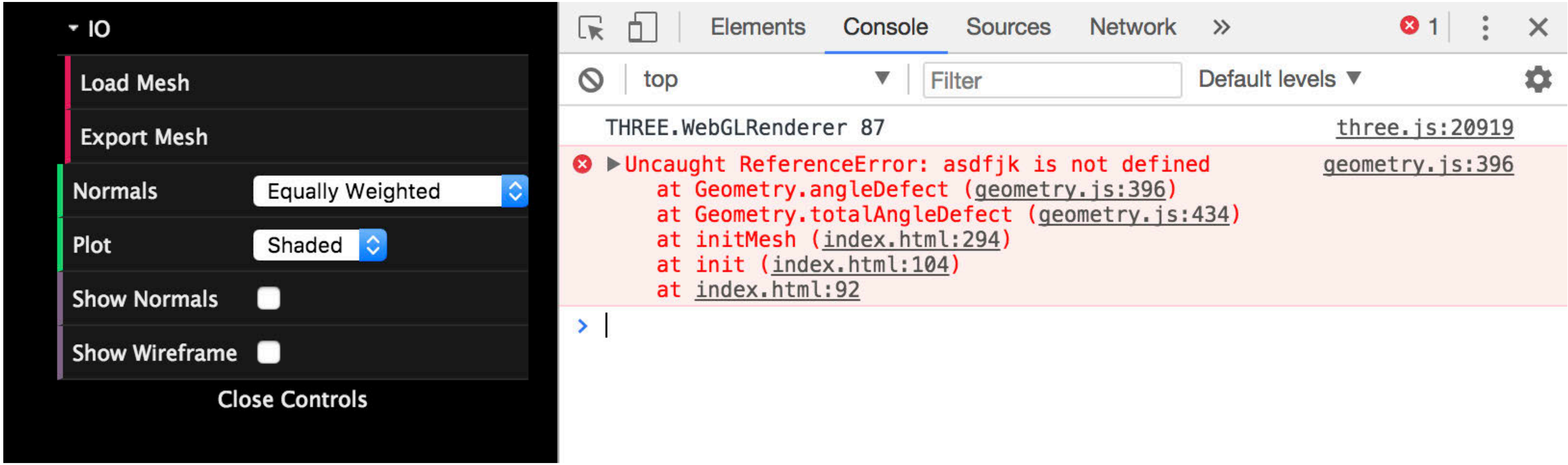
- If implemented correctly, should look like this:





Debugging in the Browser

- If it's not working, try taking a look at the debugger
- In Chrome: `View`⟶`Developer`⟶`JavaScript Console`



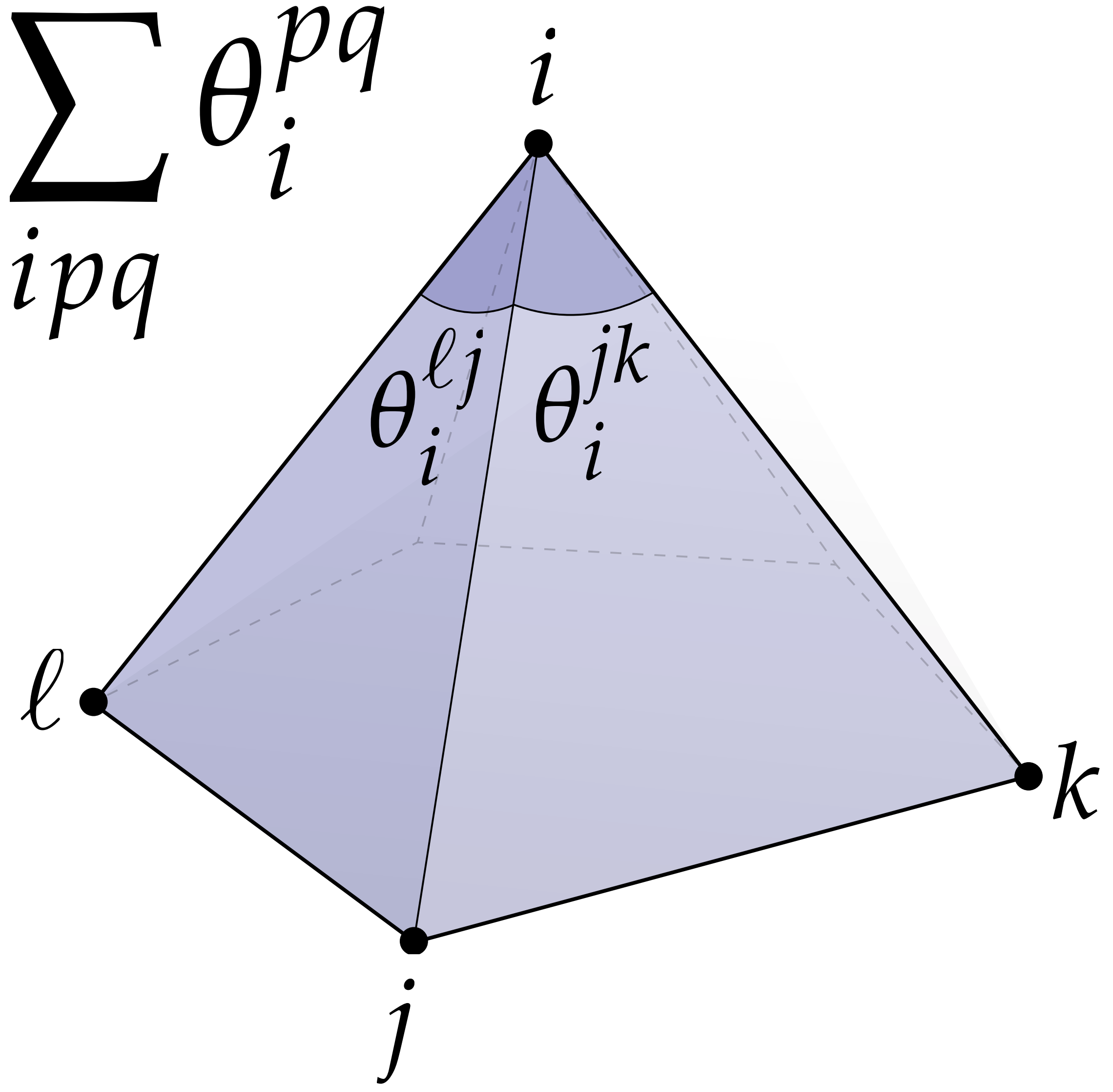
Implementing Discrete Gauss Curvature

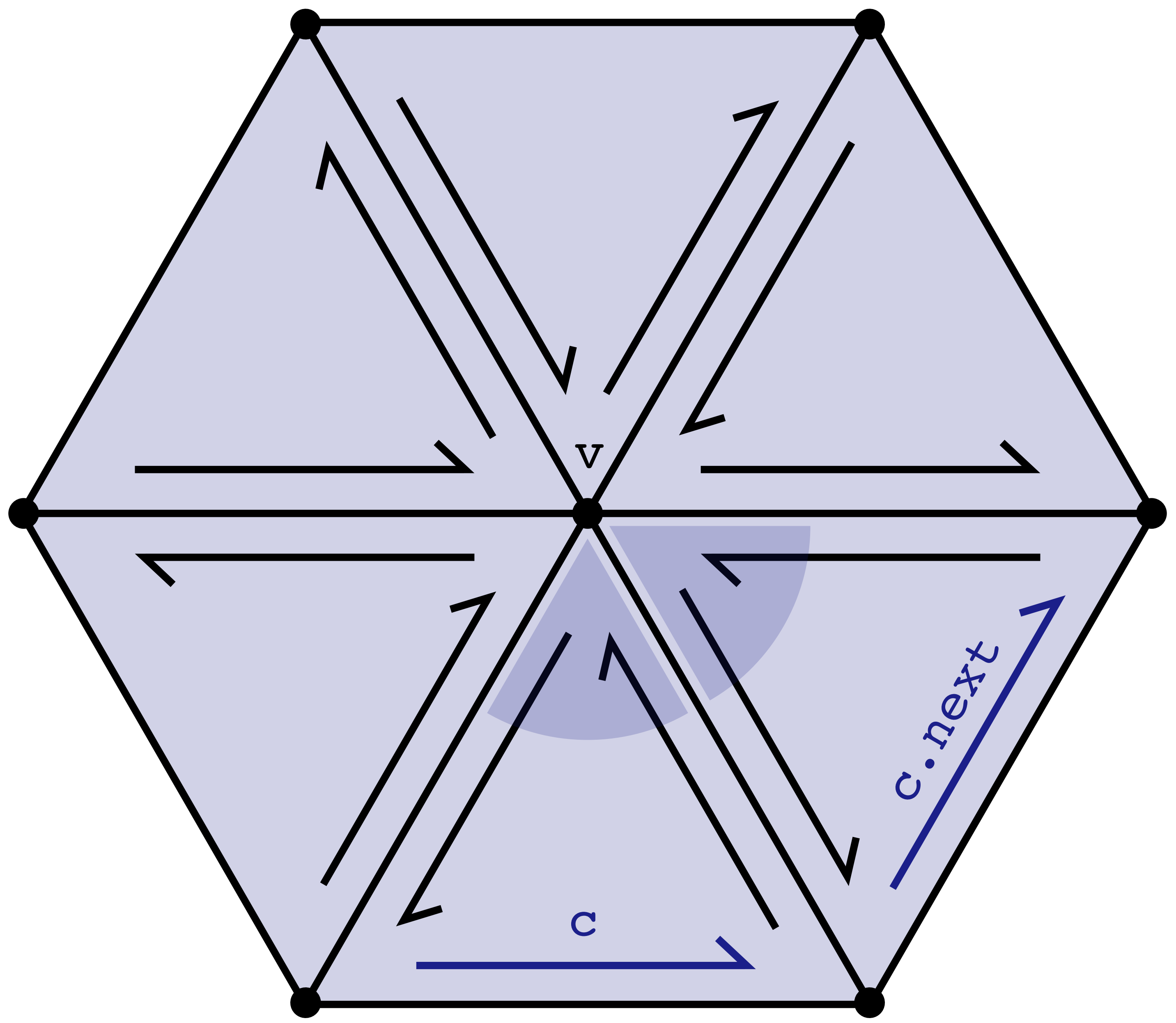
In practice, don't have to do quite so much work:

```
let angleSum = 0.0; // will accumulate sum of angles

// iterate over "corners" rather than halfedges
// (really just halfedges in disguise…)
for (let c of v.adjacentCorners()) {
    angleSum += this.angle(c); // accumulate angle sum
}

return 2.0*Math.PI - angleSum; // return defect
```

(This is the code you'll find in the reference solution.)

$$K_i := 2\pi - \sum_{ipq} \theta_i^{pq}$$



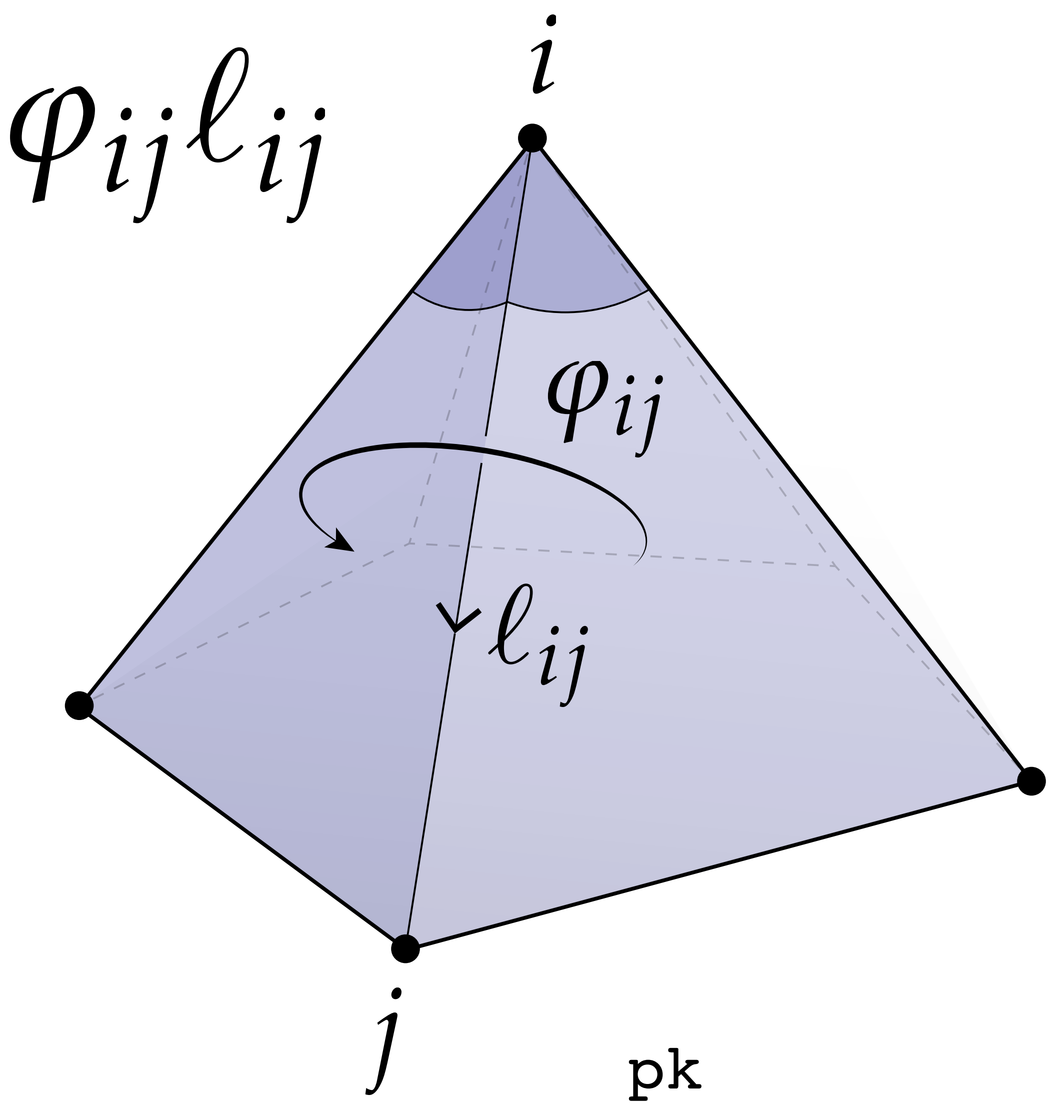


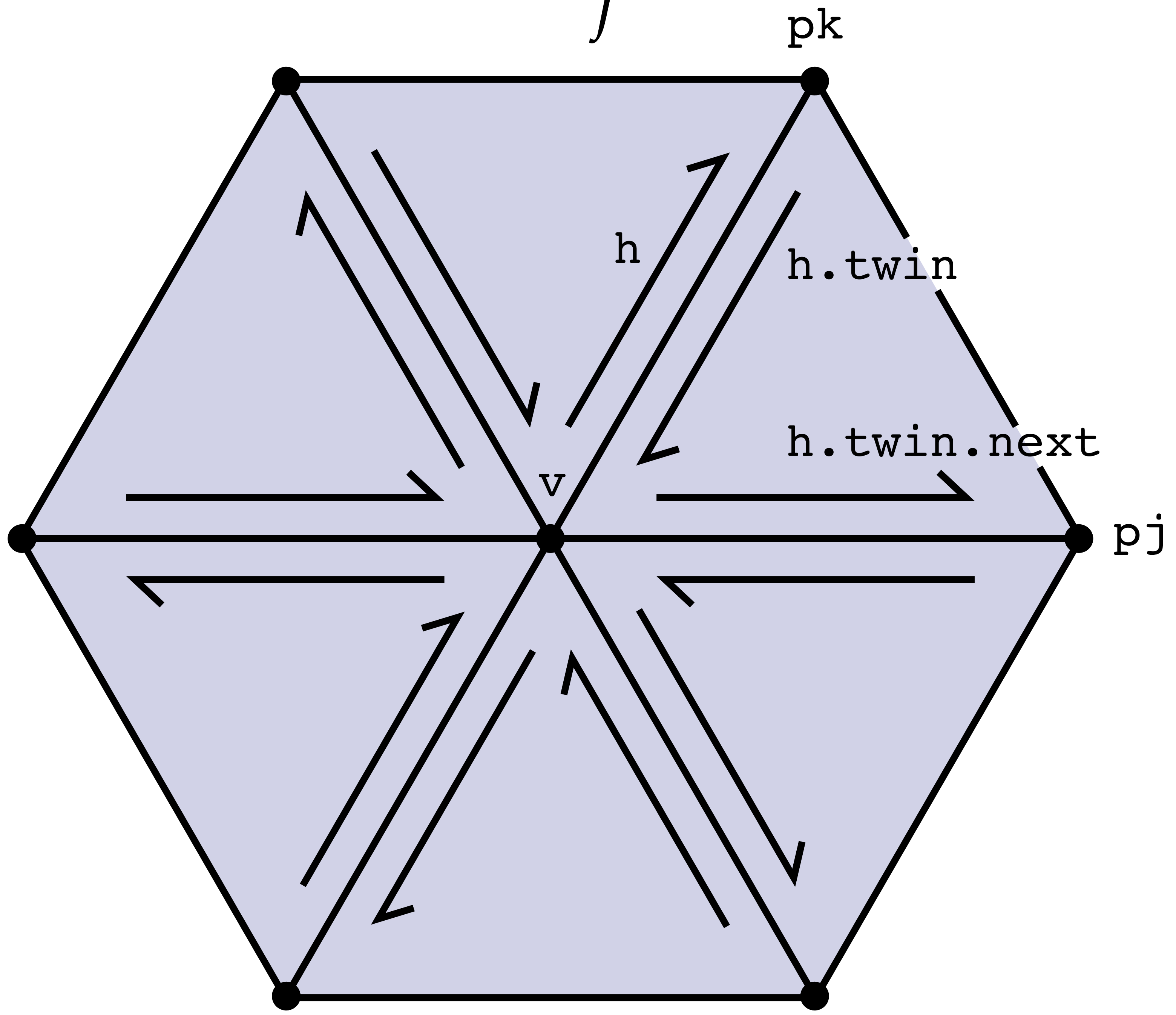
Implementing Discrete Mean Curvature

This time, start with "high level" implementation:

```
let sum = 0.0; // will accumulate sum

// iterate over outgoing halfedges
for (let h of v.adjacentHalfedges()) {
   sum += 0.25 * this.length(h.edge) * this.dihedralAngle(h);
}

return sum;
```

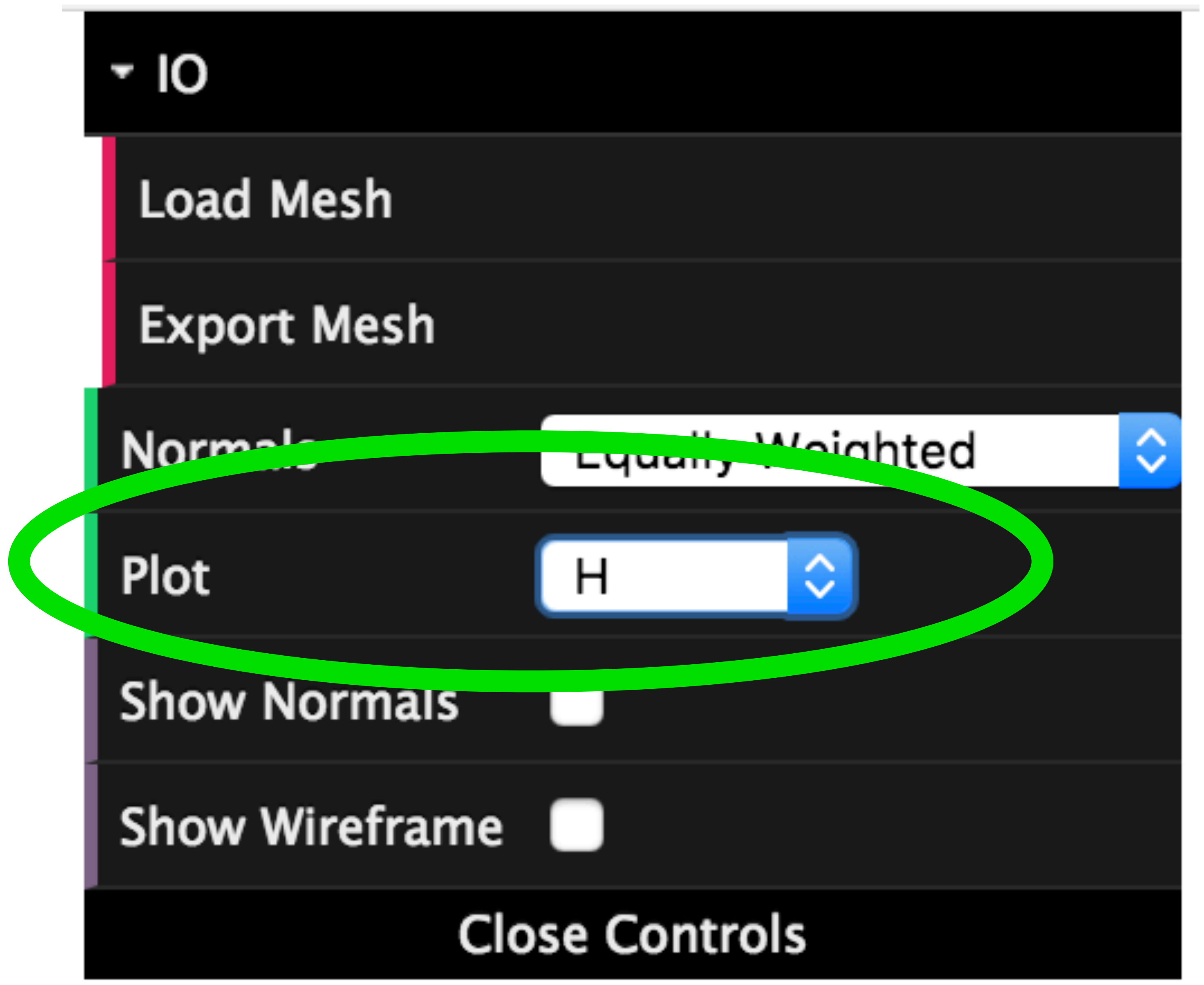
$$H_i := \frac{1}{4}\sum_{ij} \varphi_{ij}\ell_{ij}$$





Discrete Mean Curvature—Visualized

- If implemented correctly, should look like this:

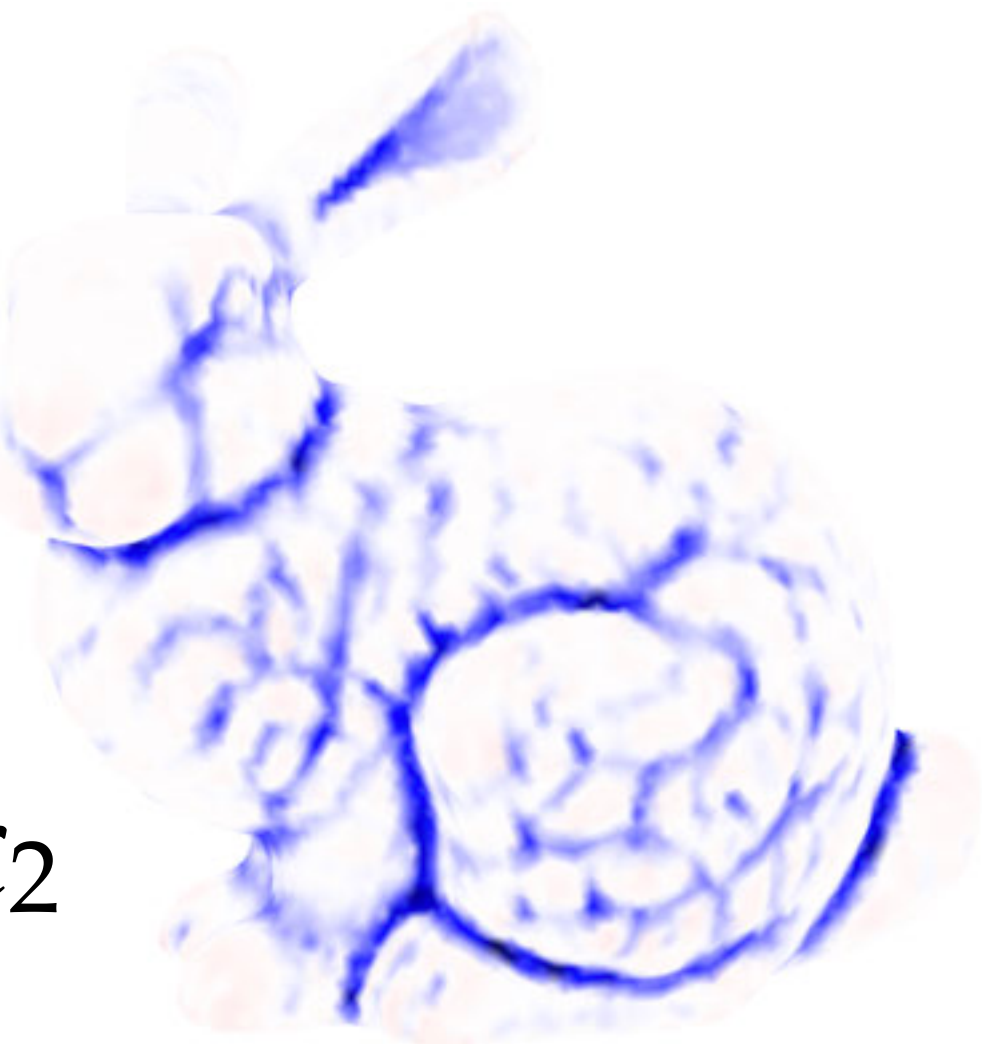


Discrete Principal Curvatures

- Given Gauss curvature K and mean curvature H, can easily solve for principal curvatures κ_1, κ_2:

$$\begin{aligned} K &= \kappa_1 \kappa_2 \\ H &= \tfrac{1}{2}(\kappa_1 + \kappa_2) \end{aligned} \quad \Longrightarrow \quad \begin{aligned} \kappa_1 &= H + \sqrt{H^2 - K} \\ \kappa_2 &= H - \sqrt{H^2 - K} \end{aligned}$$

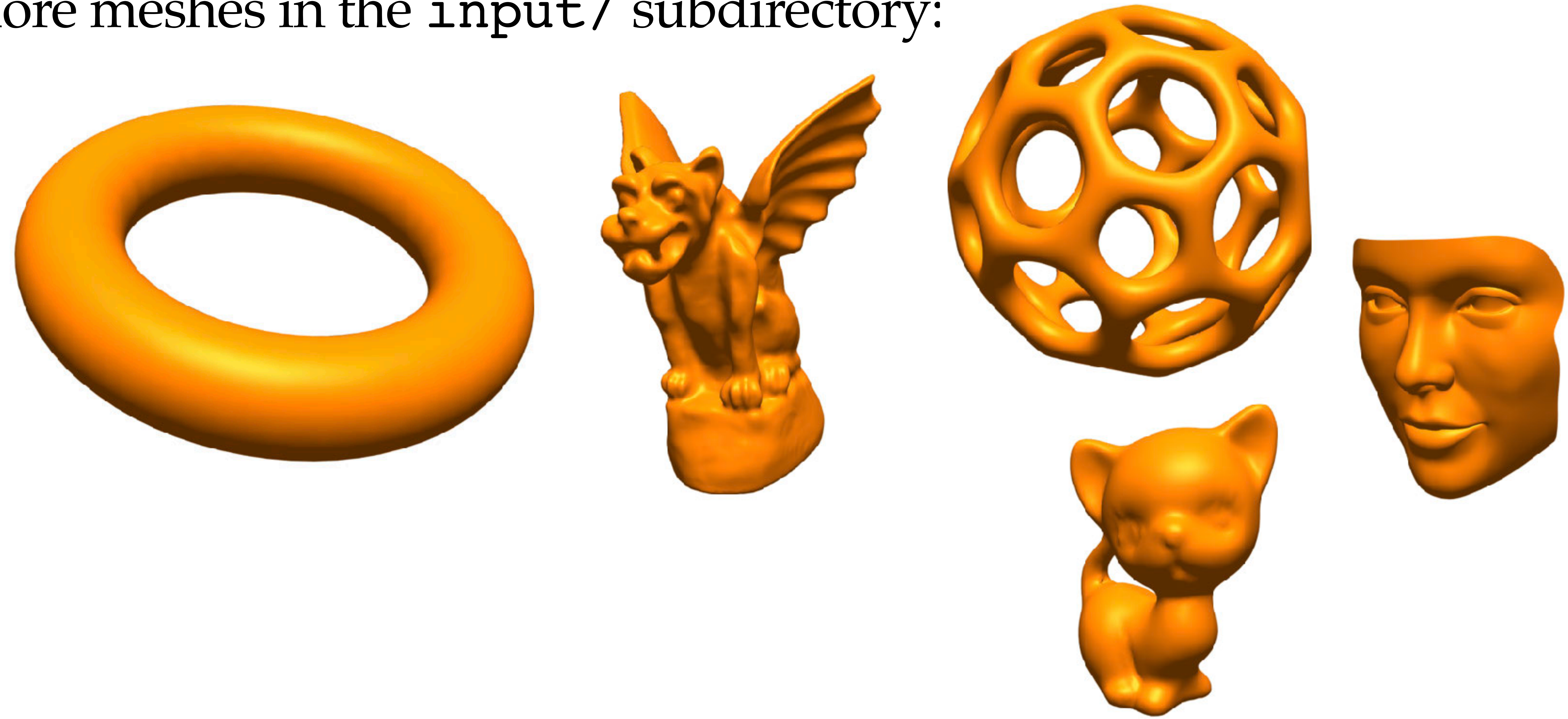
κ_1



κ_2

Try it out on some other meshes...

- More meshes in the `input/` subdirectory:

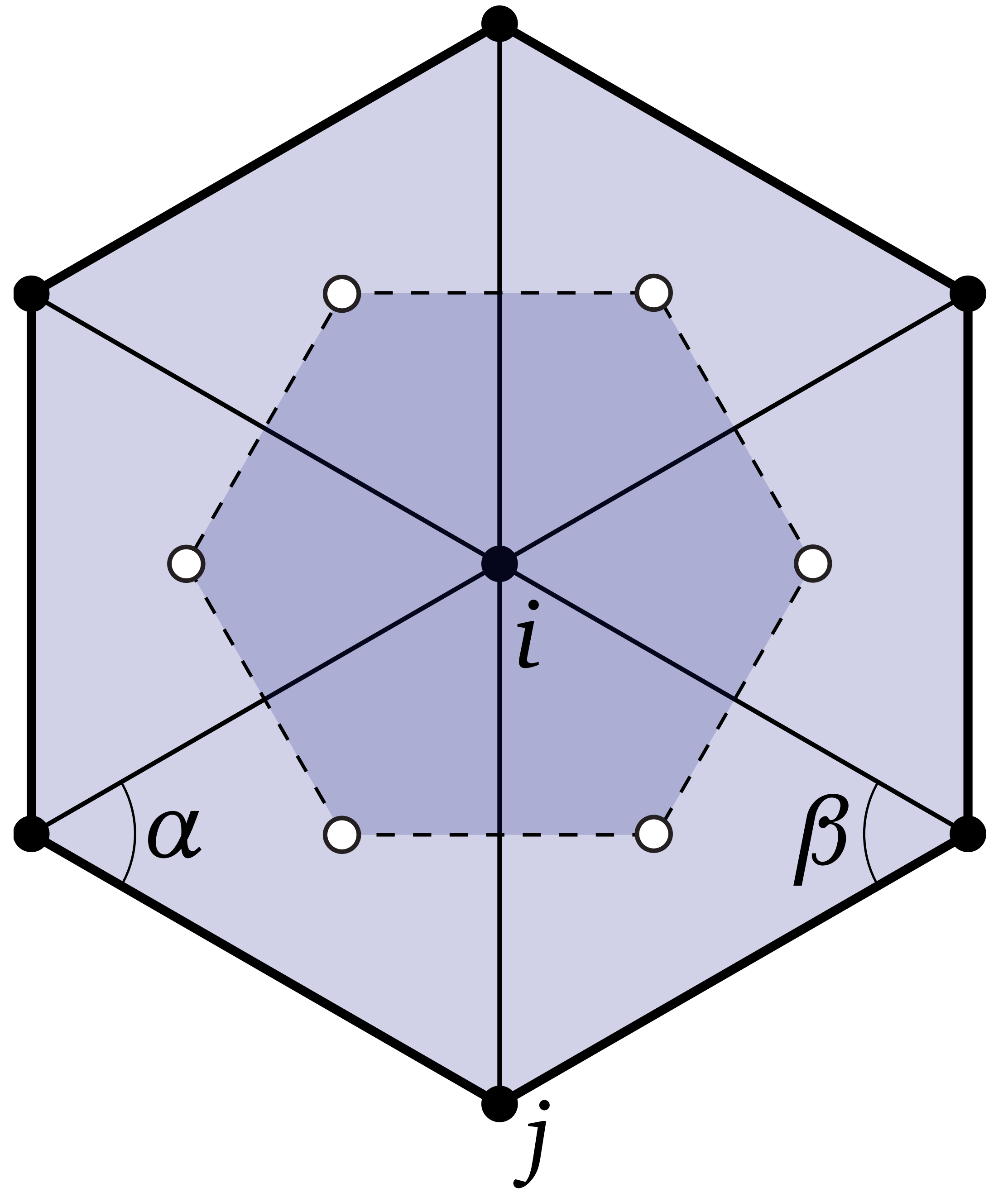


(**Q:** What do you notice about the **total** Gaussian curvature?)

Activity 2: Discrete Laplace-Beltrami

Discrete Laplace-Beltrami Operator

- Fundamental to geometry, PDEs
  - Laplace, Poisson, heat equation, wave equation…
- "Swiss army knife" of geometry processing algorithms
- Easily discretized via *cotan formula*:



$$(\Delta u)_i = \frac{1}{2A_i} \sum_{ij} (\cot \alpha_{ij} + \cot \beta_{ij})(u_j - u_i)$$

A_i — area of dual cell made by triangle circumcenters

Solving Discrete Equations

- Code for *evaluating* Laplace-Beltrami not much different from discrete curvatures
- But what about *solving* an equation, like a Poisson equation $\Delta u = f$?
- Becomes a system of linear equations (one per vertex):

$$\frac{1}{2A_i}\sum_{ij}(\cot\alpha_{ij}+\cot\beta_{ij})(u_j-u_i) = f_i$$

- To solve numerically, need to encode as a matrix equation:

$$\mathsf{Lu} = \mathsf{f}$$

$$\begin{aligned} \mathsf{L} &\in \mathbb{R}^{n\times n} \\ \mathsf{u},\mathsf{f} &\in \mathbb{R}^{n} \end{aligned}$$

n — number of vertices

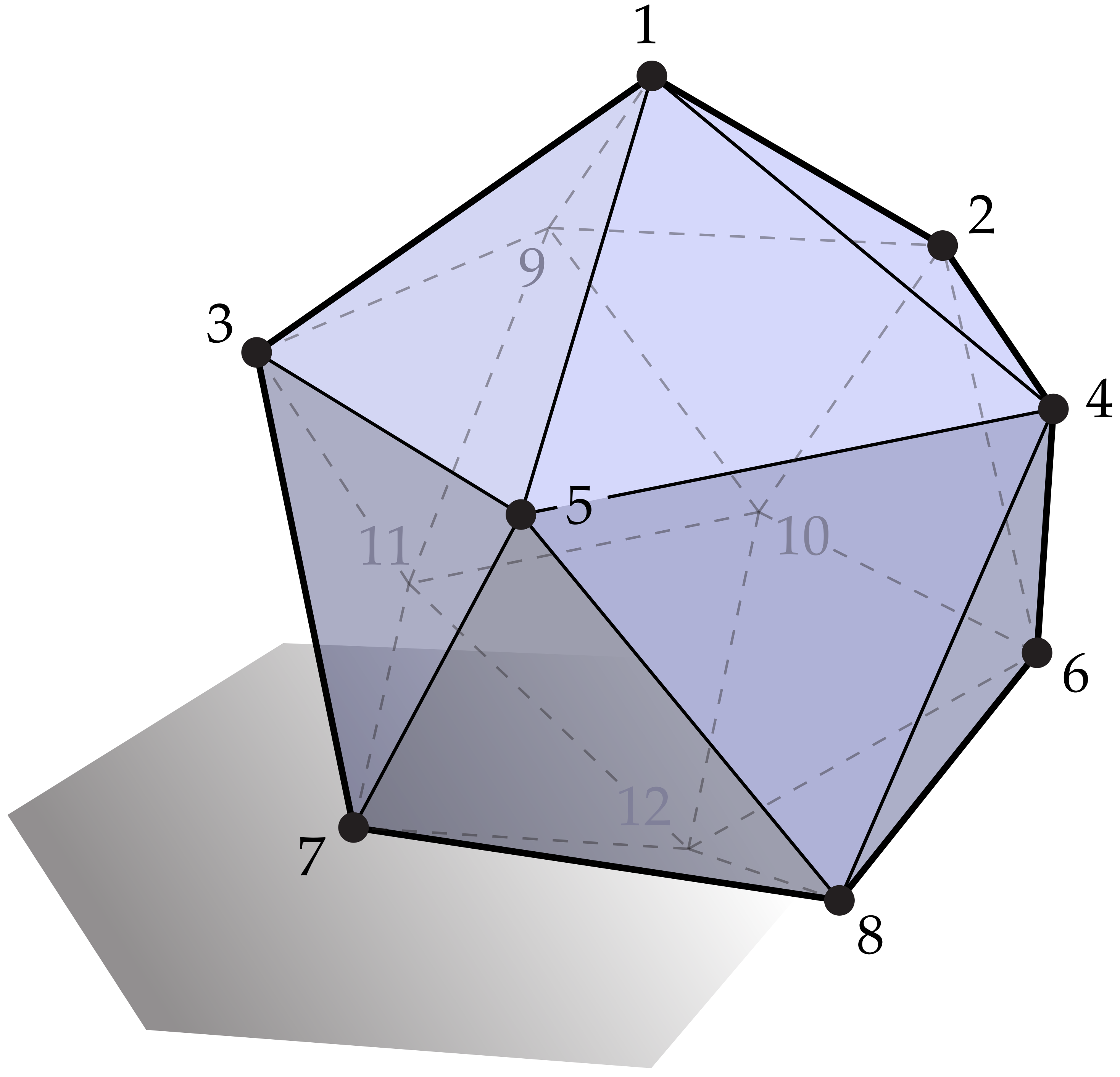
Meshes and Matrices

- To express Poisson equation in matrix form, first need to *index* the mesh
- *I.e.*, pick a bijection between the vertices and the integers 1, …, n
- Row i of matrix L encodes the linear equation corresponding to vertex i
- *E.g.*, row 5 of the Laplace matrix:

$$\begin{array}{cc} & \begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \\ 5 & \left[\begin{array}{cccccccccccc} w & 0 & w & w & -5w & 0 & w & w & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

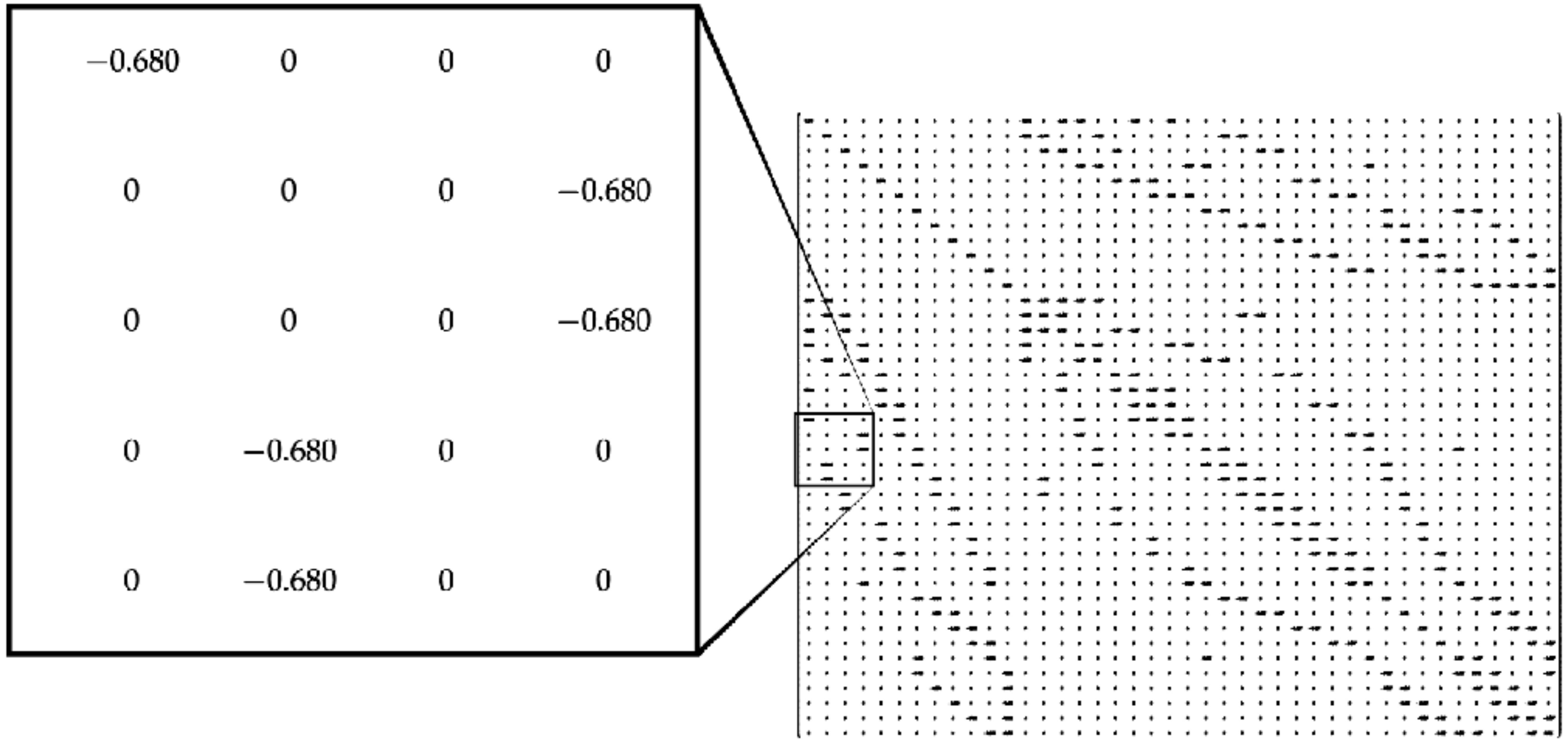
$$\mathsf{u} \in \mathbb{R}^n \quad \Longrightarrow \quad (\mathsf{Lu})_i = \sum_{ij} w_{ij}(\mathsf{u}_j - \mathsf{u}_i)$$

$(w = 2\cot(\pi/3)/A)$



Sparse vs. Dense Matrices

- For large meshes, most entries of Laplace matrix will be *zero*
- Rather than store all zeros explicitly, encode by *"sparse matrix"*
- In code: express as list of **`Triplets`**
  - **`(value, row, column)`**
  - duplicates are summed



Making the System Symmetric

- Numerically, useful to decompose Laplace matrix L into two parts:
  - *mass matrix* M — diagonal matrix of dual areas A_i
  - *stiffness matrix* C — symmetric semidefinite matrix of cotan weights

$$\frac{1}{2A_i}\sum_{ij}(\cot\alpha_{ij} + \cot\beta_{ij})(u_j - u_i) \qquad\qquad \mathsf{L} = \mathsf{M}^{-1}\mathsf{C}$$

- Can now solve Poisson equation w/ efficient solvers for symmetric systems:

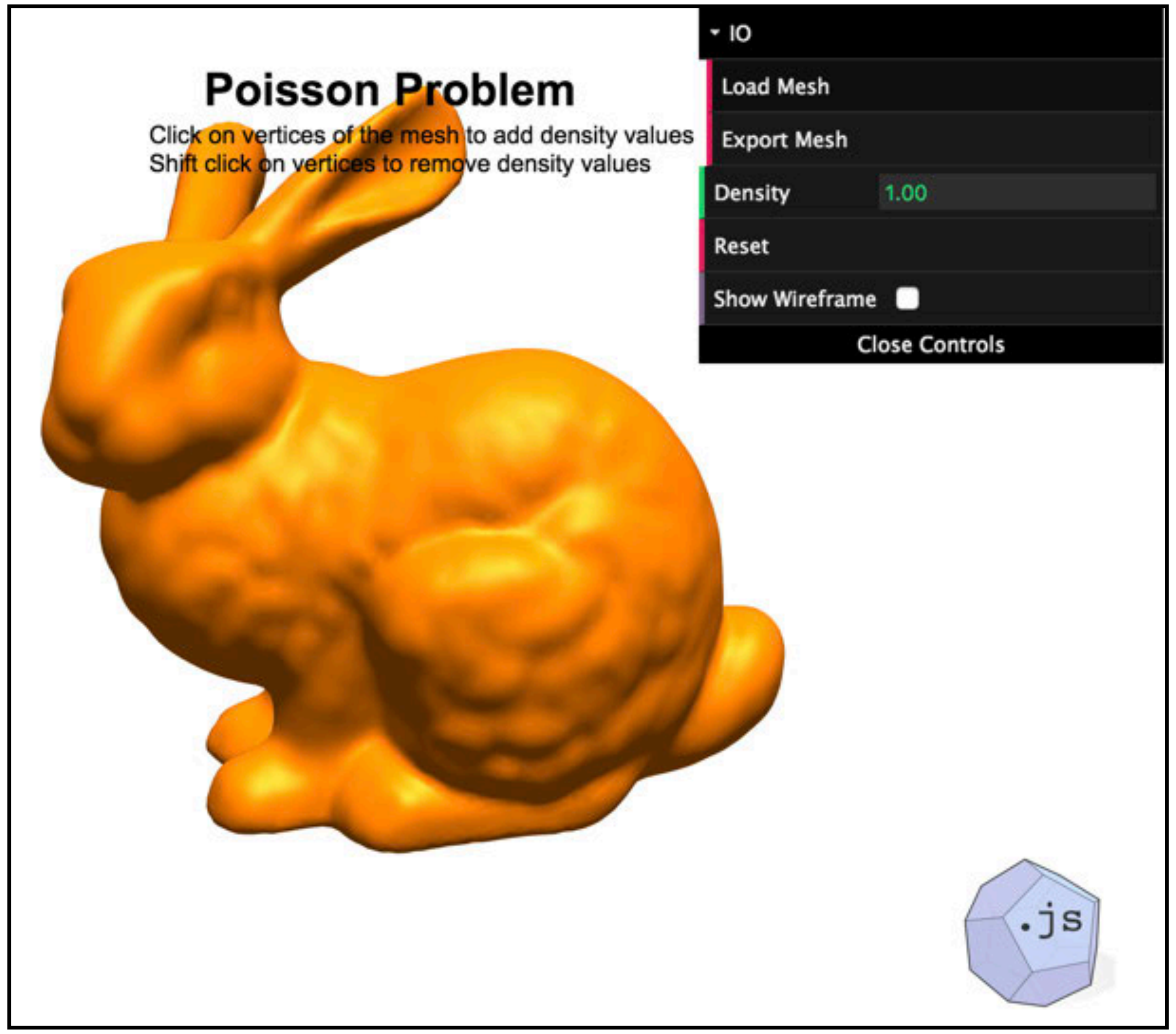
$$\mathsf{Lu} = \mathsf{f} \qquad \textit{(not symmetric)}$$

$$\iff \mathsf{Cu} = \mathsf{Mf} \qquad \textit{(symmetric)}$$

(This is how we'll do it in our code!)

Implementing a Discrete Poisson Equation

- As before, start in **`ddg-js_skeleton/`** *(we'll fill this one in)*
- Open:
  - in **web browser**: `projects/poisson-problem/index.html`
  - in **text editor**: `core/geometry.js`
- Will implement methods to build our two matrices:
  - `massMatrix(vertexIndex)`
  - `laplaceMatrix(vertexIndex)`

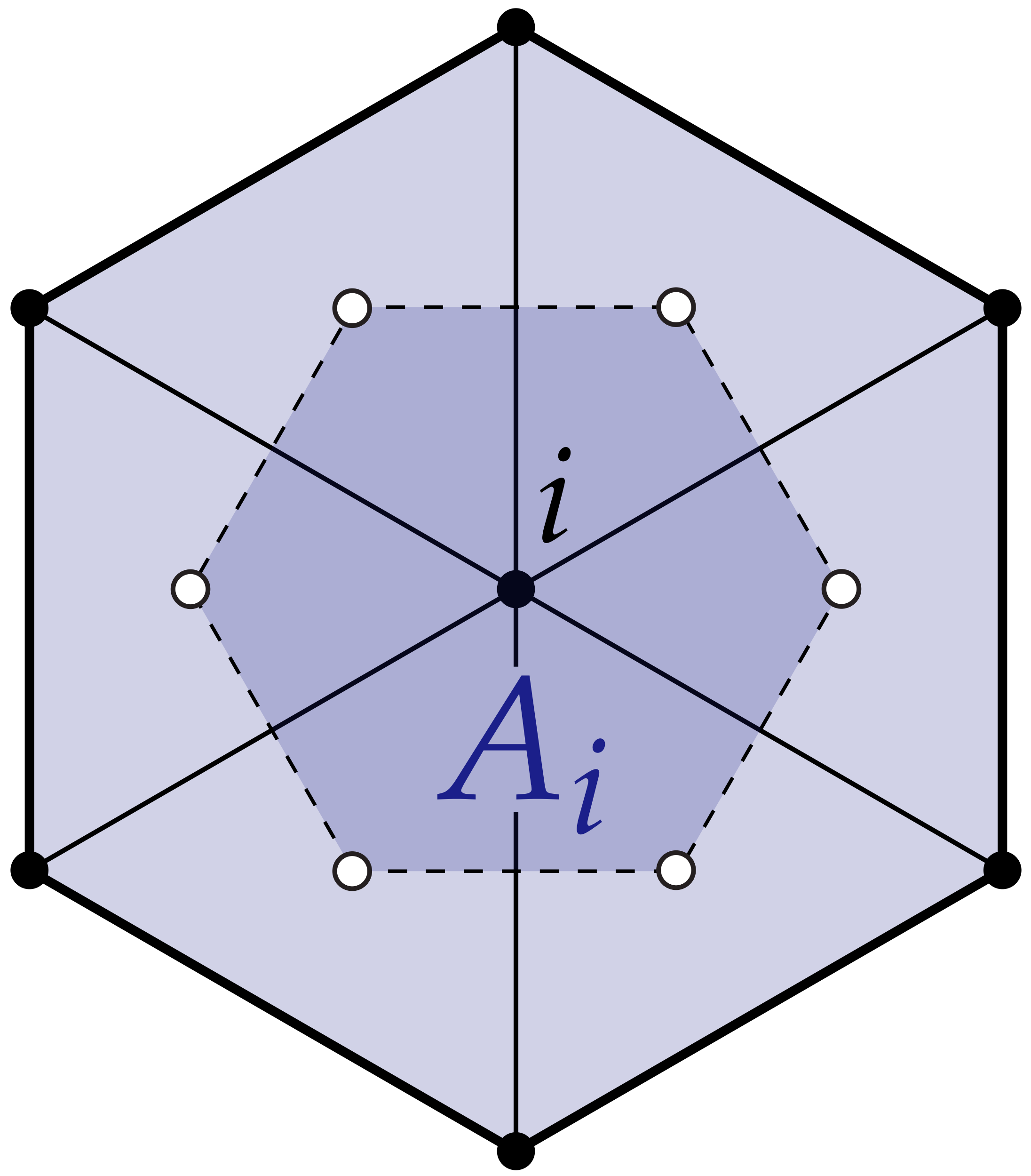


Building the Mass Matrix

Diagonal matrix; just have to build a **`Triplet`** for each vertex

```
// "vertexIndex" specifies the index of each vertex
massMatrix(vertexIndex) {
   // get the number of vertices in the mesh
   let n = this.mesh.vertices.length;

   // initialize a list of triples for an n x n matrix
   let T = new Triplet(n, n);

   // loop over all vertices
   for (let v of this.mesh.vertices) {

      // get the index of this vertex
      let i = vertexIndex[v];

      // create a triplet on the diagonal
      T.addEntry(this.circumcentricDualArea(v), i, i);
   }

   // convert list of triplets to final sparse matrix
   return SparseMatrix.fromTriplet(T);
}
```

$$\mathsf{M} = \begin{bmatrix} A_1 & & \\ & A_2 & \\ & & \ddots \end{bmatrix}$$



`circumcentricDualArea(v)`

(already implemented)

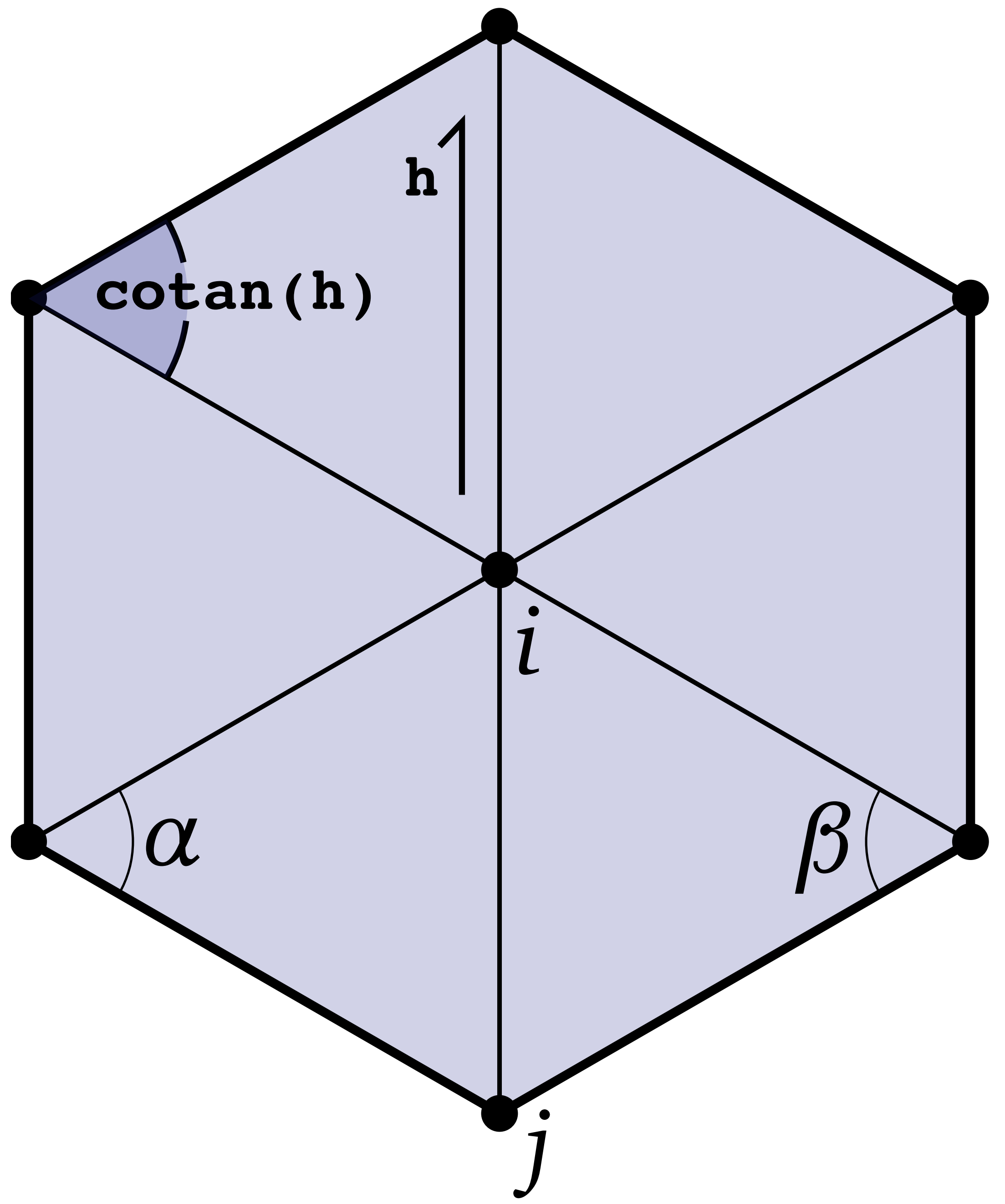
Building the Stiffness Matrix

Similar logic; now just have to loop over neighbors:

```
let n = this.mesh.vertices.length;
let T = new Triplet(n, n);
for (let v of this.mesh.vertices) {
   let i = vertexIndex[v]; // get index of this vertex
   let sum = 1e-8; // (helps w/ numerics)

   // iterate over outgoing halfedges
   for (let h of v.adjacentHalfedges()) {
      // get index of neighbor
      let j = vertexIndex[h.twin.vertex];

      // set entry Lij to cotan weight
      let weight = (this.cotan(h)+this.cotan(h.twin))/2;
      T.addEntry(-weight, i, j);

      sum += weight; // accumulate diagonal weight
   }
   T.addEntry(sum, i, i); // set entry Lii
}
// convert list of triplets to final sparse matrix
return SparseMatrix.fromTriplet(T);
```

*(make **positive** definite, for solver)*

$$\mathsf{L}_{ij} = \begin{cases} -\frac{1}{2}(\cot\alpha_{ij} + \cot\beta_{ij}), & i \neq j \\ -\sum_{j\neq i} \mathsf{L}_{ij}, & i = j \end{cases}$$



cotan(h)

(already implemented)

Solving the Poisson Equation

Now just solve linear system (already implemented in **`scalar-poisson-problem.js`**):

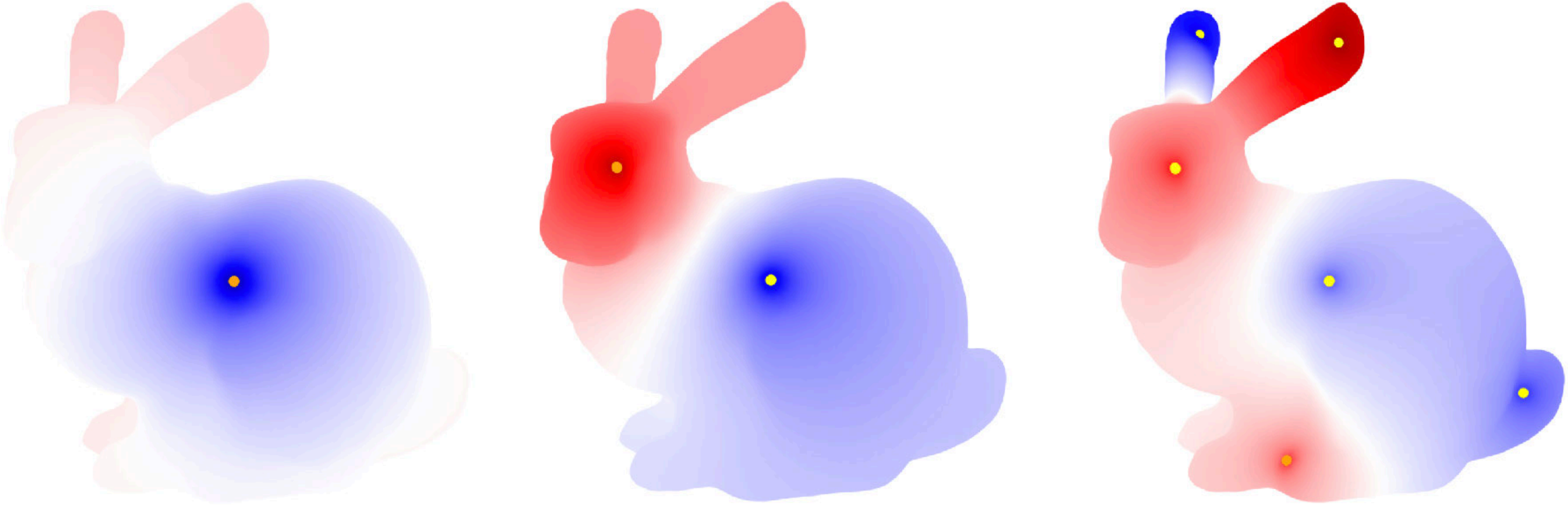
```
// index vertices
this.vertexIndex =
indexElements(geometry.mesh.vertices);

// build mass matrix
M = geometry.massMatrix(this.vertexIndex);

// build and *factor* laplace matrix
// (allows use to efficiently re-solve many
// Poisson equations w/ different right-hand sides)
C = geometry.laplaceMatrix(this.vertexIndex);
factoredC = C.chol(); // Cholesky factorization

// multiply right-hand side by mass matrix
let Mf = this.M.timesDense(f);

// solve linear system(s) using prefactored matrix
let u = factoredC.solvePositiveDefinite(Mf);

return u;
```

$$
\begin{aligned}
Cu &= Mf_1 \\
Cu &= Mf_2 \\
Cu &= Mf_3 \\
&\vdots
\end{aligned}
$$

(Most of the cost is in solving 1st equation; remaining equations are much cheaper.)

Poisson Equation on a Surface

- Solve for electrostatic potential φ corresponding to a given charge density ρ:



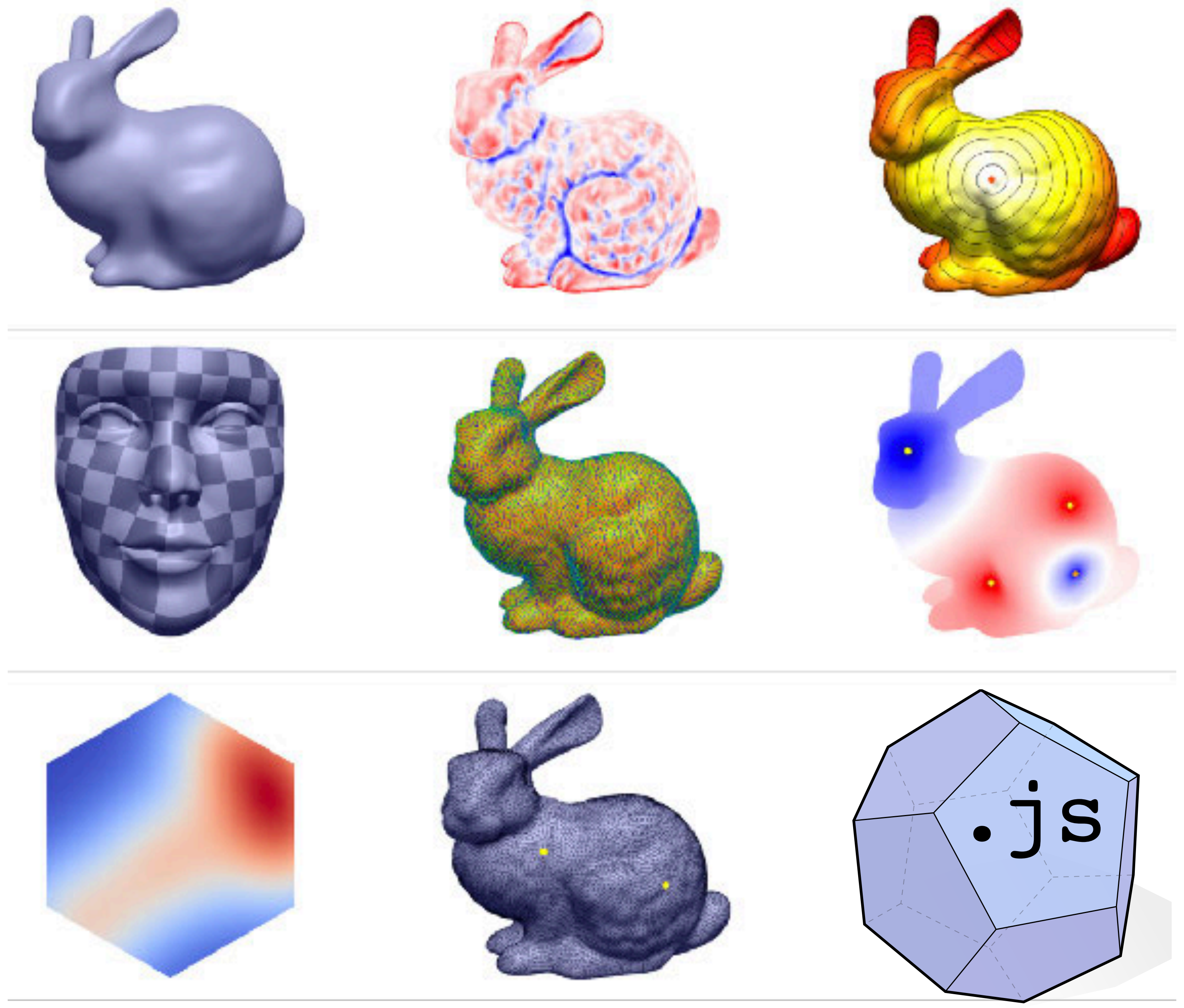
$$\Delta\varphi = \rho$$

(Open **`projects/poisson-problem/index.html`** in web browser)

Activity III: Experiment

More Algorithms

- Many more algorithms from DDG can be implemented with *surprisingly* little work beyond what you've done today
- Common pattern:
  - compute some local quantity (e.g., curvature)
  - solve a Poisson-like equation
  - possibly apply some more local computation
  - (or do something like this in an iterative loop)
- Let's take a look at some examples…



Heat & Curvature Flow

- Poisson equation is stationary solution to *heat flow* (w/source term)
- Easy to implement heat flow using same matrices:

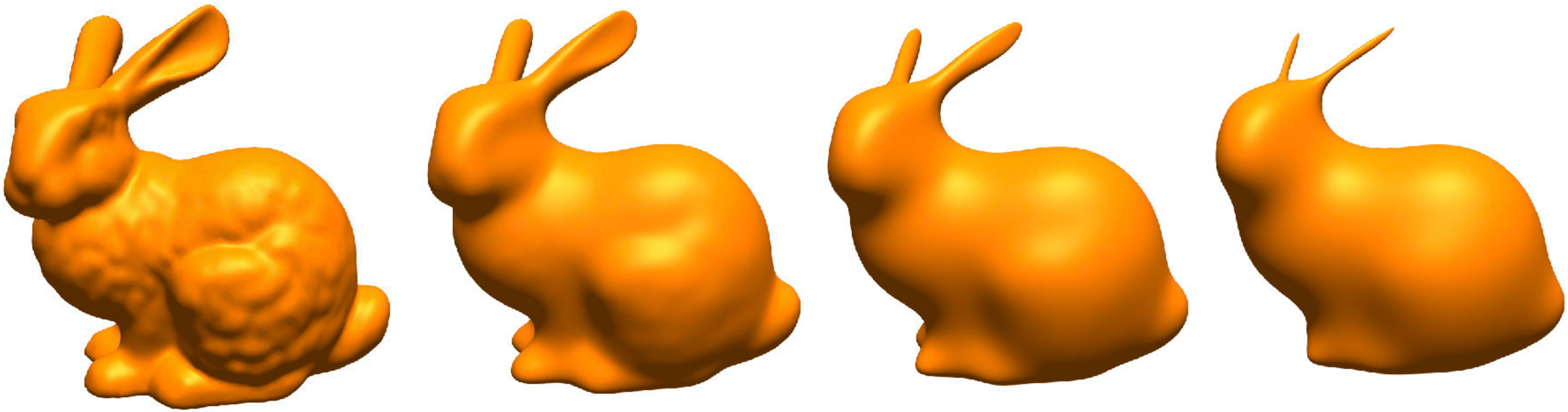
$$\frac{d}{dt}u = \Delta u \qquad \Longrightarrow \qquad (\mathsf{I} + \tau\mathsf{C})\mathsf{u}^{k+1} = \mathsf{M}\mathsf{u}^k$$

$$\begin{array}{rcl} \mathsf{I} \in \mathbb{R}^{n\times n} & \text{—} & \text{identity matrix} \\ \tau > 0 & \text{—} & \text{time step} \end{array}$$

- From here, can get *mean curvature flow* by making two small changes:
  - replace u with vertex positions of surface (i.e., surface immersion)
  - update matrices M and C every time the surface changes

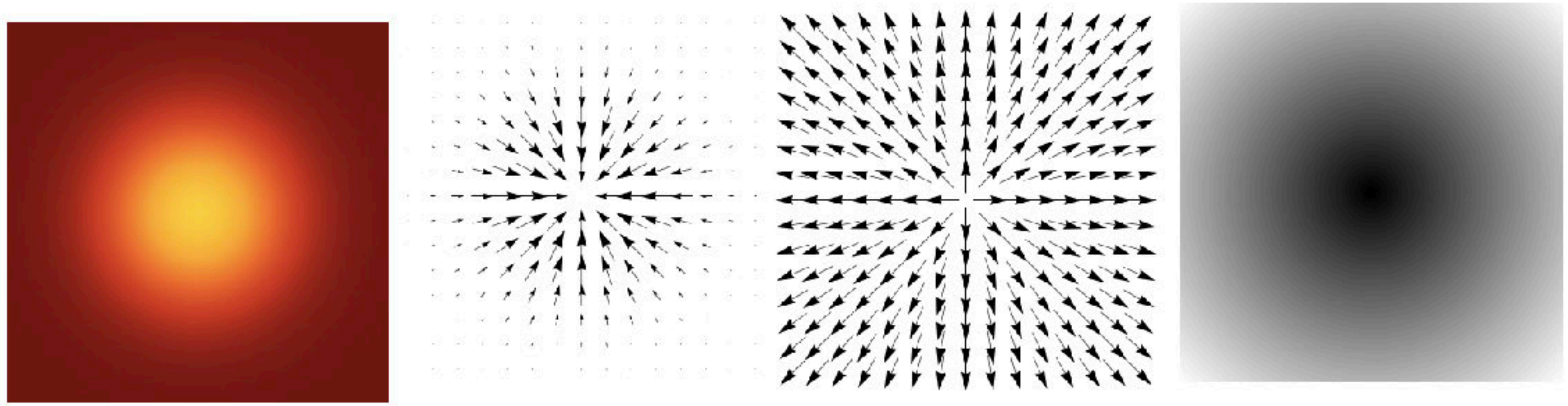
Mean Curvature Flow

Run from **`ddg-js_solutions/projects/geometric-flow/index.html`**



Geodesic Distance

- Can also find geodesic distance to one (or more) source points via a familiar pattern:



u $\quad$ ∇u $\quad$ X $\quad$ ϕ

Algorithm 1 The Heat Method

I. Integrate the heat flow $\dot{u} = \Delta u$ for some fixed time t.

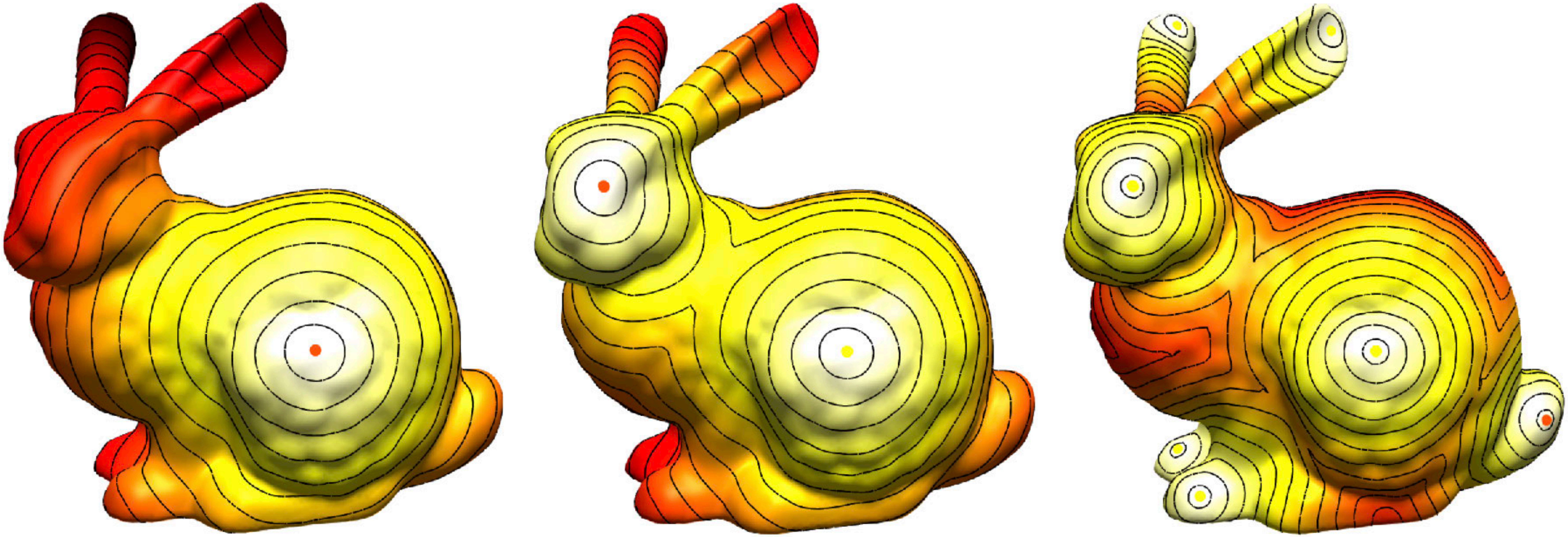
II. Evaluate the vector field $X = -\nabla u / |\nabla u|$.

III. Solve the Poisson equation $\Delta \phi = \nabla \cdot X$.

for details, see Crane, Weischedel, & Wardetzky, *"The Heat Method for Distance Computation"* (2017)

Geodesic Distance

Run from **`ddg-js_solutions/projects/geodesic-distance/index.html`**



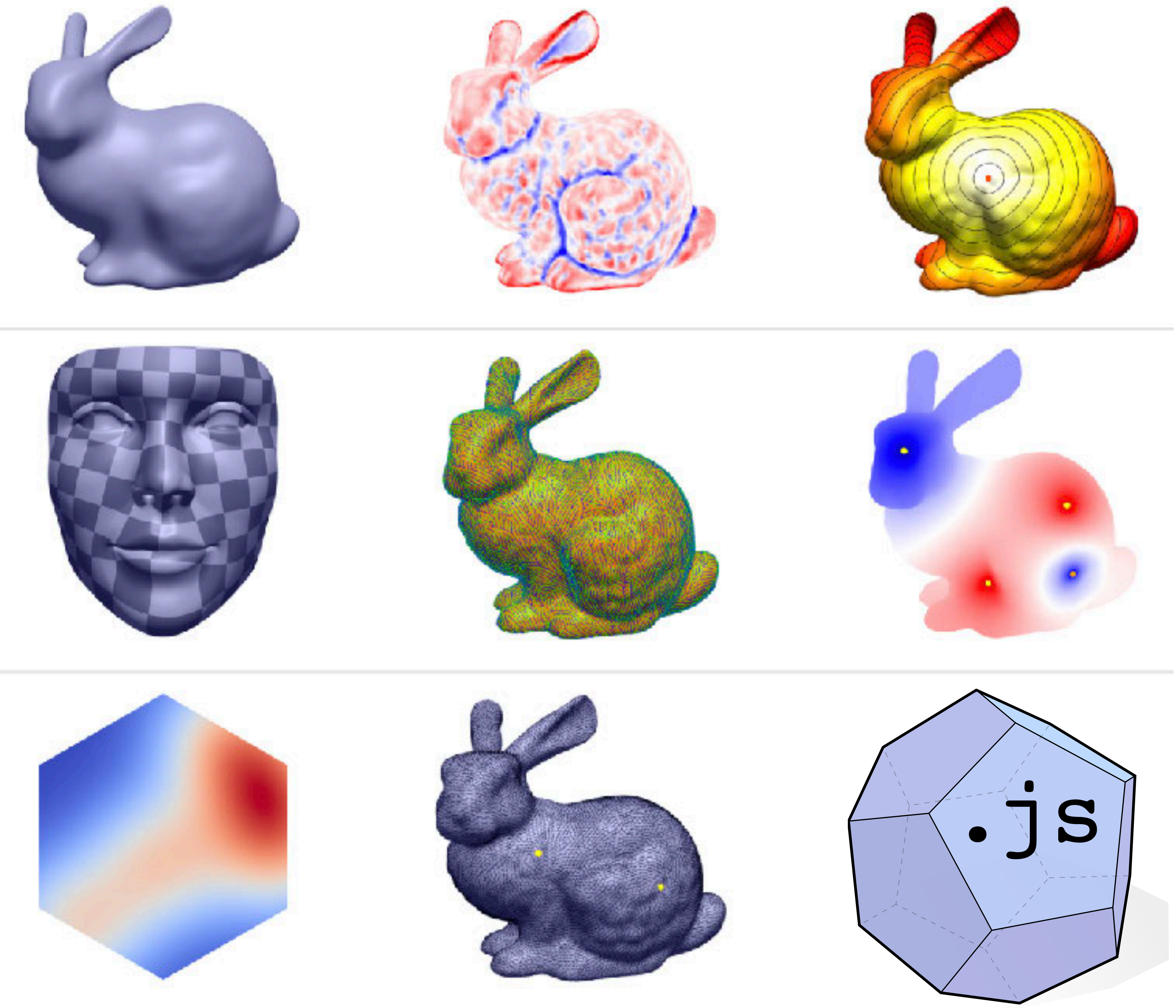
for details, see Crane, Weischedel, & Wardetzky, *"The Heat Method for Distance Computation"* (2017)

Additional Examples

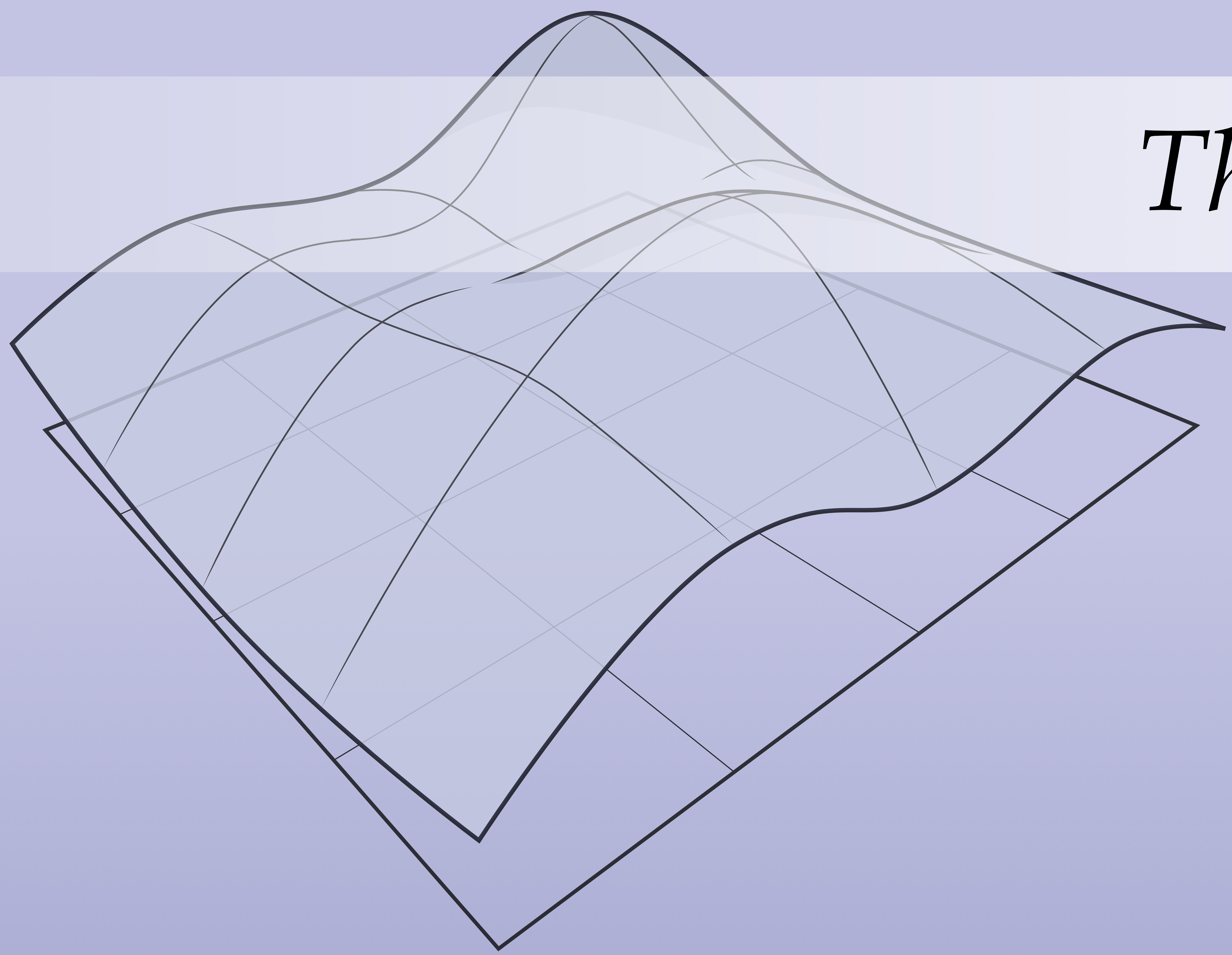
- *Many* more algorithms boil down to solving Poisson-like equations via cotan-Laplace matrix:
  - conformal flattening (**`parameterization`**)
  - Helmholtz-Hodge decomposition (**`vector-field-decomposition`**)
  - direction field w/ prescribed singularities (**`direction-field-design`**)
  - optimal transport
  - shape descriptors
  - ...
- Take a look at examples in **`ddg-js_solutions/projects/`**

Thanks!

- Thanks for participating! Let us know if you have any questions…



(See also: http://geometry.cs.cmu.edu/ddg)



Thanks!

AMS SHORT COURSE

DISCRETE DIFFERENTIAL GEOMETRY

Joint Mathematics Meeting • San Diego, CA • January 2018