Geometric Modeling
Assignment 2: Implicit Surface Reconstruction

Acknowledgements: Olga Diamanti, Julian Panetta
HW 2: Surface Reconstruction

• **Input:** point cloud with normals

• **Output:** smooth surface mesh passing near each point

• **How?** Find out tomorrow from Daniele’s lecture! (Or keep listening…)
Implicit Surface Reconstruction

- Remember: **surface representation matters**!

- Implicit representation bypasses many headaches an explicit approach would encounter.

- Guarantees by construction:
  - 2-Manifold
  - No holes (watertight)
  - Robust to noisy point clouds
Simplifies the Problem

Surface interpolation

Implicit Rep

Scalar field interpolation

?
Constructing the Scalar Field

• Interpolate information from the input cloud:
  
  • **Points** tell us where the zero level set should go:
    
    \[ f(p_i) = 0 \]
  
  • **Normals** define (locally) inside/outside
Step 1: Build the Constraint Set

- Point constraints $f(p_i) = 0$ are insufficient (trivial solution $f = 0$)
- Incorporate normal info with additional off-surface constraints:
  
  $$f(p_i + \epsilon n_i) = \epsilon$$
  $$f(p_i - \epsilon n_i) = -\epsilon$$
Step 2: Construct Interpolant

- Construct regular grid
- Compute nodal scalar field satisfying constraints (approximately).

- Method: **MLS**
  (Moving Least Squares)
Interpolation Problem

- List of 3N constraint locations, \( \mathbf{c}_i \) (e.g. \( \mathbf{p}_0, \mathbf{p}_0 + \epsilon \mathbf{n}_0, \ldots \))

- List of 3N values, \( d_i \)

- Together, they describe 3N constraints of the form \( f(\mathbf{c}_i) = d_i \)

- **Goal:** find the “best” \( f \) in the span of chosen basis functions \( \mathbf{b}(\mathbf{x}) \):

\[
f(\mathbf{x}) = \sum_j b_j(\mathbf{x}) a_j
\]

(By tuning weights \( a_j \) to best approximate constraints)
Basis Functions

• For this assignment, we’ll use **polynomial basis functions**:

(but in 3D)
Constraints in the Basis

• We can express our constraints in this basis:

\[ f(c_i) = \sum_j b_j(c_i)a_j = d_i \]

In matrix form:

\[ B \mathbf{a} = \mathbf{d} \]

• Where matrix \( B_{ij} := b_j(c_i) \)
  (columns hold basis function’s value at every constraint location).

\[
B = \begin{bmatrix}
1 & x_1 & y_1 & \cdots \\
1 & x_2 & y_2 & \cdots \\
\vdots & \vdots & \vdots & \ddots 
\end{bmatrix}
\]
Overconstrained Linear System

• We’ll have many more constraints than basis functions…

• Least-squares solution?

\[
\min_f \sum_i (f(c_i) - d_i)^2
\]

\[
\min_a \|Ba - d\|^2
\]

• What’s bad about this?
Problems with Least-squares

- **Global problem:** large matrices (even if basis functions are local)
- **Need many, high-degree basis functions**
  - Evaluating interpolant becomes expensive
- Better idea:
  - Construct **low degree, local interpolants** and stitch them together
Moving Least Squares (MLS)

- MLS builds distinct local interpolant around every eval pt!
- But final stitched function is still guaranteed smooth.
- Idea: weight the constraints based on distance to eval pt $x$:

$$f_x := \underset{f}{\text{argmin}} \sum_i w(||x - c_i||) (f(c_i) - d_i)^2$$

- **Constraints with zero weight disappear!**
  (Choose weight function so few kept $\Rightarrow$ small linear system)
MLS in Matrix Form

$$\min_f \sum_i w(\|x - c_i\|)(f(c_i) - d_i)^2$$

$$\min_a \|Ba - d\|^2_{W(x)}$$

$$\|Ba - d\|^2_{W(x)} := (Ba - d)^T W(x)(Ba - d)$$

$$W(x) = \begin{bmatrix} w(\|x - c_1\|) \\ \vdots \\ w(\|x - c_{3N}\|) \end{bmatrix}$$

Note: some papers call this $W(x)^2$
MLS Coefficients, Closed Form

- MLS objective function is quadratic in coefficients $a$; find optimum by differentiating and solving a linear system:

$$0 = \nabla_a \left( (B a - d)^T W(x) (B a - d) \right)$$

$$= 2 B^T W(x) B a - 2 B^T W(x) d$$

- Thus the coefficients for point $x$ are given by solving the system:

$$\left( B^T W(x) B \right) a(x) = B^T W(x) d$$

for $a(x)$. 
Step 2: Construct Interpolant

- Finally, fill in the grid!
- Evaluate local MLS interpolant at each grid point $\mathbf{x}$.

$$f_{\mathbf{x}}(\mathbf{x}) = \sum_j b_j(\mathbf{x}) a_j(\mathbf{x})$$
Wendland Weights

• You’ll use Wendland weights for $w$ in this assignment

• Vanish at dist “$h$” from eval pt (most constraints disappear)

$$w(r) := \begin{cases} 
(1 - \frac{r}{h})^4 \left(4 \frac{r}{h} + 1\right) & \text{if } r < h \\
0 & \text{otherwise}
\end{cases}$$
Step 3: Extract Zero Level Set

- Use the **marching cubes** algorithm to extract the grid function’s zero isosurface

- Just call igl::copyleft::marching_cubes
Marching Cubes: General Idea

• Look up triangles to create in each grid cell based on corner values:
Final Result from Marching Cubes
Provided Code

- Implements pipeline but uses analytic signed distance fn for sphere in place of MLS
NanoGUI

• IGL Viewer uses NanoGUI: http://nanogui.readthedocs.io/

• You’ll need to add widgets to configure additional variables.
NanoGUI: Adding Settings

• Thankfully, this is really easy:

```cpp
viewer.callback_init = [&](Viewer &v) {
    // Add widgets to the sidebar.
    v.ngui->addGroup ("Reconstruction Options");
    v.ngui->addVariable("Resolution", resolution);
    v.ngui->addButton ("Reset Grid", [&](){
        // Recreate the grid
        createGrid();
        // Switch view to show the grid
        callback_key_down(v, '3', 0);
    });

    // Add more parameters to tweak here...
    v.screen->performLayout();
    return false;
};
```

• (C++ lambda expressions)
Provided Example: Implicit Sphere

• Step 1: Compute an axis-aligned bounding box

```cpp
/**************** createGrid() **********/
// Grid bounds: axis-aligned bounding box
Eigen::RowVector3d bb_min, bb_max;
bb_min = P.colwise().minCoeff();
bb_max = P.colwise().maxCoeff();

// Bounding box dimensions
Eigen::RowVector3d dim = bb_max - bb_min;
```
Provided Example: Implicit Sphere

• Step 2: construct a grid over the bounding box

```cpp
/***/
// Grid spacing
const double dx = dim[0] / (double)(resolution - 1);
const double dy = dim[1] / (double)(resolution - 1);
const double dz = dim[2] / (double)(resolution - 1);
// 3D positions of the grid points -- see slides or marching_cubes.h for ordering
grid_points.resize(resolution * resolution * resolution, 3);
// Create each gridpoint
for (unsigned int x = 0; x < resolution; ++x) {
    for (unsigned int y = 0; y < resolution; ++y) {
        for (unsigned int z = 0; z < resolution; ++z) {
            // Linear index of the point at (x,y,z)
            int index = x + resolution * (y + resolution * z);
            // 3D point at (x,y,z)
            grid_points.row(index) = bb_min + Eigen::RowVector3d(x * dx, y * dy, z * dz);
        }
    }
}
```
Provided Example: Implicit Sphere

• Step 3: Fill grid with the values of the implicit function

```cpp
/** evaluateImplicitFunc() ***/
// Scalar values of the grid points (the implicit function values)
grid_values.resize(resolution * resolution * resolution);

// Evaluate sphere’s signed distance function at each gridpoint.
for (unsigned int x = 0; x < resolution; ++x) {
    for (unsigned int y = 0; y < resolution; ++y) {
        for (unsigned int z = 0; z < resolution; ++z) {
            // Linear index of the point at (x,y,z)
            int index = x + resolution * (y + resolution * z);
            // Value at (x,y,z) = implicit function for the sphere
            grid_values[index] = (grid_points.row(index) - center).norm() - radius;
        }
    }
}
```

**f(x) = ||x - c|| - r**
Provided Example: Implicit Sphere

• Step 4: run marching cubes

```cpp
igl::copyleft::marching_cubes(grid_values, grid_points, resolution, resolution, resolution, V, F);
```

input: implicit function values at grid points
Provided Example: Implicit Sphere

• Step 4: run marching cubes

```cpp
igl::copyleft::marching_cubes(grid_values, grid_points, resolution, resolution, resolution, V, F);
```

input: grid point positions
Provided Example: Implicit Sphere

• Step 4: run marching cubes

```cpp
igl::copyleft::marching_cubes(grid_values, grid_points, resolution, resolution, resolution, V, F);
```

input: grid size (x, y, z)
Provided Example: Implicit Sphere

• Step 4: run marching cubes

\[
\text{igl::copyleft::marching_cubes(grid\_values, grid\_points, resolution, resolution, resolution, V, F)}
\]

output: vertices and faces
Bonus: Better Normal Constraints

• Our method implemented only point constraints
• Normals “constrained” using inward- and outward-offset value constraints
  • Leads to undesirable surface oscillation
• Solution: use the normal to define a linear function at each sample point; interpolate these functions with MLS.

Bonus: Better Normal Constraints

- Recall, we computed our interpolant by solving:
  \[ \min_a \|Ba - d\|^2_{W(x)} \]
  with constraint value \( d_i \) for the 3N constraint locations.

- **New scheme**: use just one constraint per sample pt

- Replace \( d_i \) with: \( s_i(x) = (x - p_i) \cdot n_i \)

- \( s_i \) is the linear function computing signed distance to \( p_i \)'s tangent plane

- Note: \( \nabla_x s_i = n_i \)
Bonus: Poisson Reconstruction

• Explicitly a fit scalar function’s gradient to the normals.
  • Smooth out sampled normals to create a global vector field \( \vec{V} \)
  • Find scalar function \( \chi \) whose gradient best approximates this vector field: \( \min_{\chi} \| \nabla \chi - \vec{V} \| \)

• Advantages:
  • No spurious sheets far from the surface!
  • Robust to noise

Questions?