Tips and Tricks for Lazy Meshing

Getting Started

Melanie Ganz-Benjaminsen
Kenny Erleben
Department of Computer Science

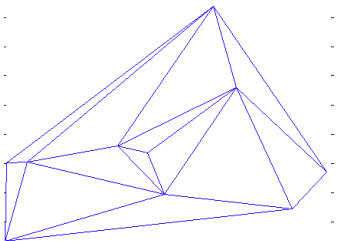
University of Copenhagen

Delaunay Triangulation

Matlab got Delaunay triangulation (both in 2D and 3D)

```
1    X = rand(10,1);
2    Y = rand(10,1);
3    T = delaunay(X,Y);
4    triplot(T,X,Y);
```

Results in

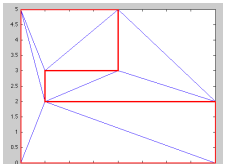


Constrained Delaunay Triangulation

Matlab got Delaunay triangulation (both in 2D and 3D)

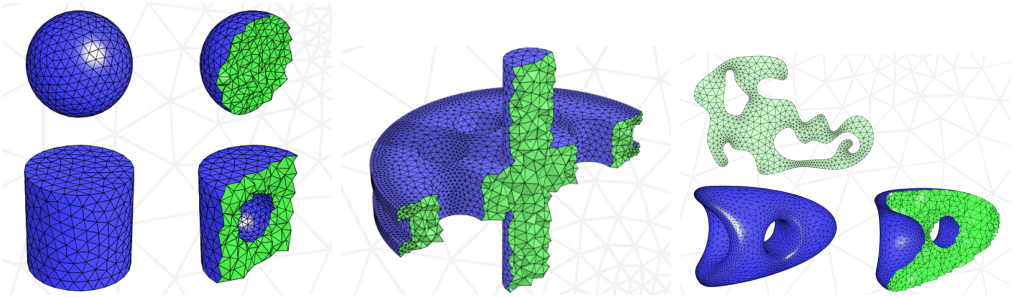
```
P = [0 0; 16 0; 16 2; 2 2; 2 3; 8 3; 8 5; 0 5];
C = [1 2; 2 3; 3 4; 4 5; 5 6; 6 7; 7 8; 8 1];
T = DelaunayTri(P, C);
triplot(T);
hold on;
plot(P(C'),P(C'+size(P,1)),'-r', 'LineWidth', 2);
hold off;
```

Results in



DistMesh - A Simple Mesh Generator in MATLAB

From `http://persson.berkeley.edu/distmesh/`

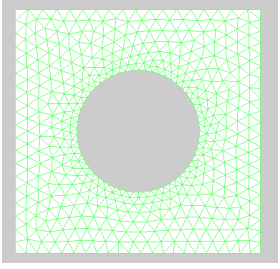


DistMesh - Example

Using

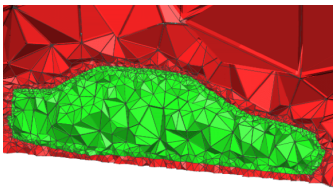
```
fd = inline('ddiff(drectangle(p, -1, 1,-1, 1),dcircle(p,0,0,0.5 ))');
fh = inline('min(4*sqrt(sum(p.^2, 2)) -1, 2)','p');
[p,t] = distmesh2d(fd,fh,0.05,[ -1,-1;1,1],[-1,-1;-1,1;1,-1;1,-1]);
```

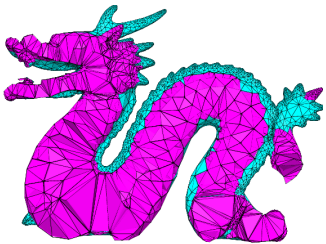
Results in

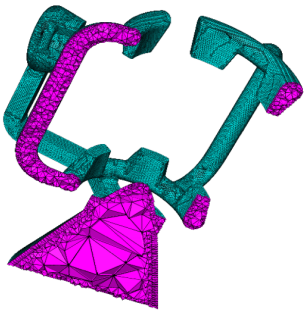


TetGen

From `http://wias-berlin.de/software/tetgen/`

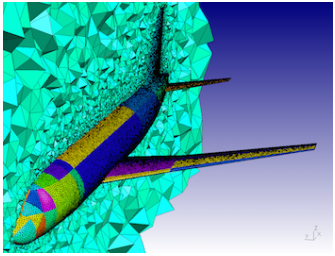


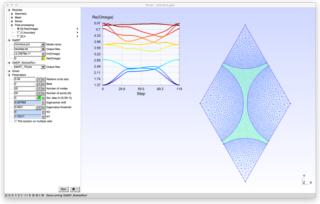


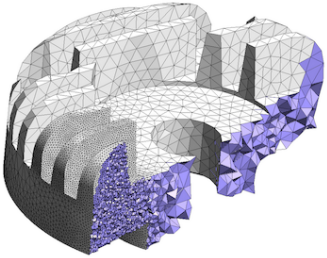


Gmsh

From `http://gmsh.info/`





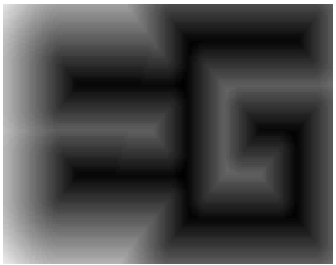


Create a Signed Distance Map

Writing

```
I = read_bw( 'EG_WEB_logo.jpg');
phi = bw2phi( I );
imagesc(phi);
```

Converts left image into right image

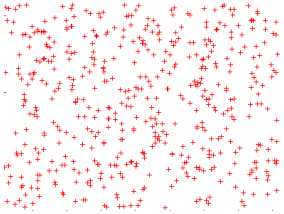


Create Random Particles

Writing

```
[M N] = size(phi);
Y = rand(1,K)*(M-6) + 3;
X = rand(1,K)*(N-6) + 3;
plot(X,Y,'r+')
```

Creates a bunch of particles

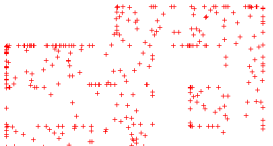


Project Particles

Writing

```
1       [GX,GY] = meshgrid( 1:N,1:M);
2       d  = interp2(GX,GY,phi,X,Y);
3       [dx,dy] = gradient(phi);
4       nx = interp2(GX,GY,dx,X,Y);
5       ny = interp2(GX,GY,dy,X,Y);
6       dx = d.*nx; dy = d.*ny;
7       X(d>0) = X(d>0) - dx(d>0);
8       Y(d>0) = Y(d>0) - dy(d>0);
```

Particles are now only inside object

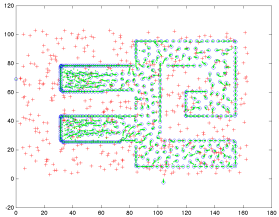


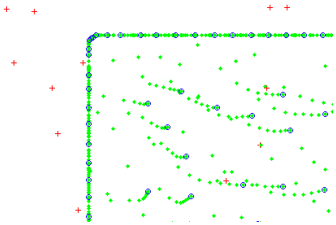
Project Particles

Particles are not placed nice! So

- Spread particles using a mass spring system (or some other physical simulation)
- Project particles again to make sure they are kept inside

Repeat until you are satisfied





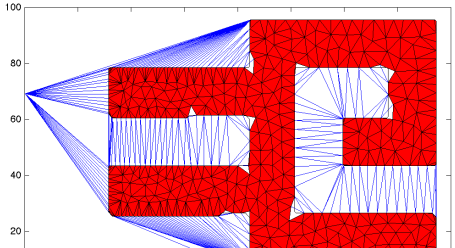
Get The Triangles

First use Delaunay triangulation

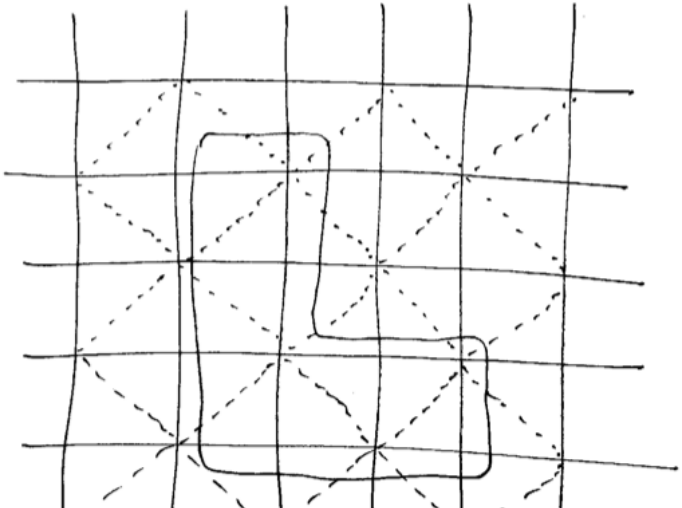
- Make ten random points inside each triangle
- If all 10 random points per triangle are inside the actual shape then keep the triangle, else remove the triangle

Now we got some red triangles representing our object

Note this is just one method in which you could extract the triangles representing your object

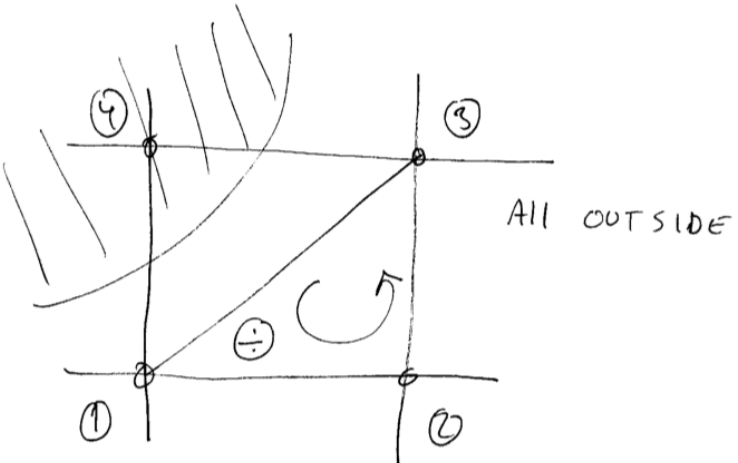


Marching Triangles – Make a Grid



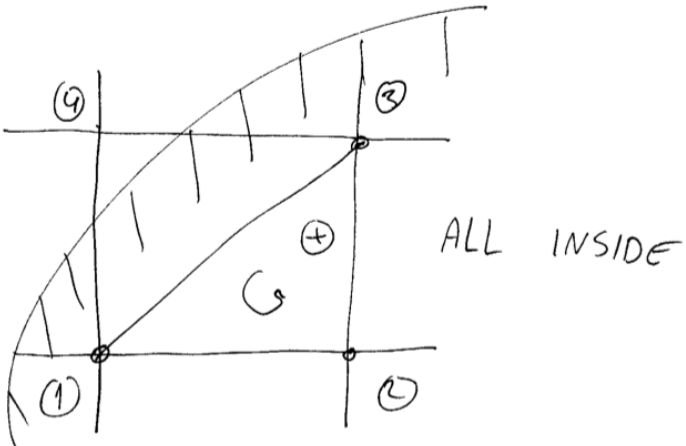
Marching Triangles – Case 1

Check if any of the edges intersect with my shape, if all are outside remove the triangle



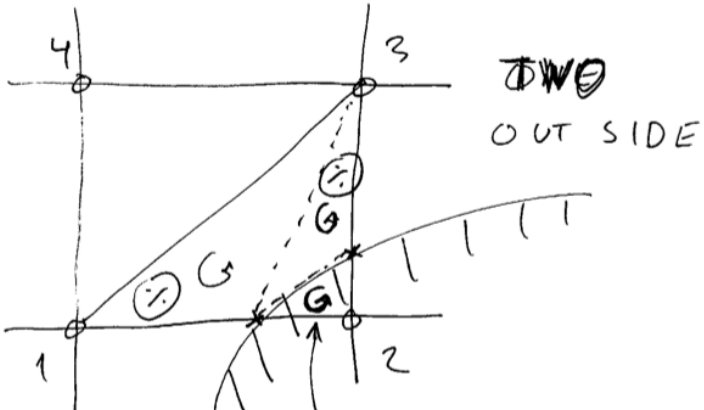
Marching Triangles – Case 2

If all are inside, keep the triangle



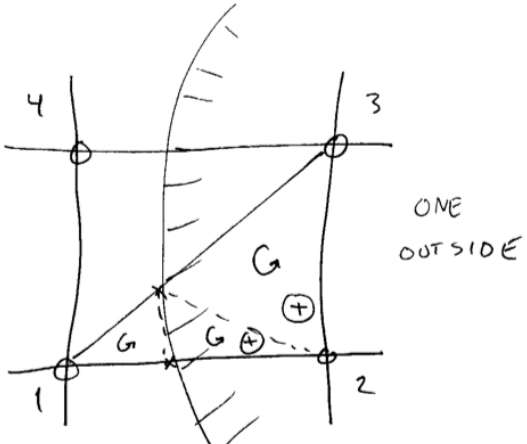
Marching Triangles – Case 3

Keep the vertices 1 and 3 only and move the vertices towards the shape by projection/interpolation



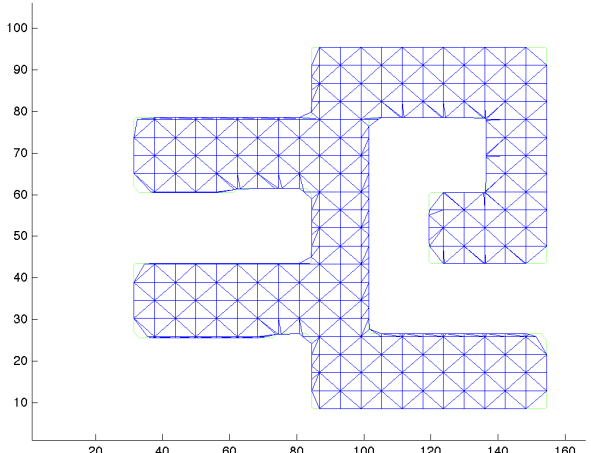
Marching Triangles – Case 4

Similar to case 3 but within the shape



Marching Triangles – Final Result

Using the rule book described above you can then arrive at the final solution depicted here



Implementing Marching Triangles

Simply described one just loop over all triangles and investigate which case each triangle belongs too. Once this is know one can generate the output triangles trivially.

However, here's some implementation advice in Matlab that we just wish to uncover to accelerate implementation tasks.

Define a Grid

Let us uncover some Matlab implementation ideas for Marching Triangles. First we define a regular grid.

```
I        = 5;   % Number of nodes along x-axis
J        = 5;   % Number of nodes along y-axis
T        = [];  % Array of output triangles
width    = 10;
height   = 10;
dx       = width/(I-1);
dy       = height/(J-1);
[GX,GY]  = meshgrid( 0 : dx : width, 0 : dy : height);
X        =  GX(:);   % X coordinates of grid nodes
Y        =  GY(:);   % Y coordinates of grid nodes
```

Notice we store the coordinates in 1D index form and not as a 2D array..

Split Grid Cells into Triangle Pattern

Next we will split the grid into triangles. The (*1) code is just for debugging, we will replace it later on.

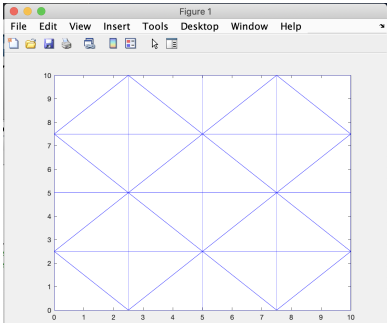
```
for i=1:I-1
  for j=1:J-1
      % Get 1D indices of cell vertices
      v1 = (i-1)*J+(j-1)+1; v2 = i*J     +(j-1)+1;
      v3 = i*J     +j     +1; v4 = (i-1)*J+j     +1;
      flip = mod( mod(i,2) + mod(j,2), 2);
      if(flip==1) % Determine triangle pattern of cell
        t = [ v1 v2 v3; v1 v3 v4];
      else
        t = [ v4 v1 v2; v2 v3 v4];
      end
      for k=1:2 % Let us output the grid triangles
          T = [ T; t(k,:) ]; % (*1)
      end
  end
end
```

The Grid Triangle Pattern

We can now visualize the grid triangles we created

```
figure();
triplot(T,X,Y)
```

This results in a figure like this



Creating An Inisde/outside Mask

We are given a black and white image as input that we convert to a signed distance field and then we compute the values of the field at the grid nodes

```
phi = bw2phi( I );
[M N]    = size(phi);
[IX,IY] = meshgrid(1:N,1:M);
d   =   interp2(IX,IY,phi,X,Y);
```

Next we wish to classify each of the triangle vertices as being inside or outside. We create a binary mask for this just before we `k`-loop,

```
    mask = (d(t)<threshold);
```

Remember that `t` holds indices of the cell triangles in the current cell.

Create a Stencil to Classify Triangles

The binary inside/outside mask of the triangle corners can now be used to create a "stencil" that uniquely identifies how to deal with each cell triangle. Like this,

```
1    for k=1:2,
2      stencil = mask(k,1) + mask(k,2)*2 + mask(k,3)*4;
3      ....
4    end
```

Observe that the stencil is converting all possible inside-outside cases into a bitmask or phrased differently a unique stencil index that tell us exactly what rule to apply to "cut" up the current triangle.

Observe that the stencil ranges from 0 to 7. It has 8 cases, but our rule book only included 4 cases.

Reducing Cases

Notice cyclic permutation of indices turns stencil cases into the same rule cases.

```
1   if stencil == 0  % All outside
2     apply_rule = 0;
3   elseif stencil == 1  % First vertex inside
4     apply_rule = 1;
5   elseif stencil == 2  % Second vertex inside
6     apply_rule = 1; t(k,:) = [t(k,2) t(k,3) t(k,1) ];
7   elseif stencil == 4  % Third vertex inside
8     apply_rule = 1;  t(k,:) = [t(k,3) t(k,1) t(k,2)];
9   elseif stencil == 3  % First and second vertices inside
10    apply_rule = 2;
11  elseif stencil == 5  % First and third vertices inside
12    apply_rule = 2; t(k,:) = [t(k,3) t(k,1) t(k,2)];
13  elseif stencil == 6  % Second and third vertices inside
14    apply_rule = 2; t(k,:) = [t(k,2) t(k,3) t(k,1)];
15  elseif stencil == 7 % All vertices inside
16    apply_rule = 3;
17  end
```

Apply the Rules

Now we can just apply the rules to create triangles

```
      xA = ?; % Do your own thing
      yA = ?; % Do your own thing
      xB = ?; % Do your own thing
      yB = ?; % Do your own thing
      if(apply_rule==3)
        T = [T; t(k,:)];
      elseif(apply_rule==2)
        X = [X' xA xB]'; Y = [Y' yA yB]';
        i3 = length(X); i2 = i3-1;
        T = [T; [t(k,1) t(k,2) i3 ]];
        T = [T; [t(k,2)     i2   i3 ]];
      elseif(apply_rule==1)
        X = [X' xA xB]'; Y = [Y' yA yB]';
        i3 = length(X); i1 = i3-1;
        T = [T; [t(k,1)   i1  i3 ]];
      end
```

Afterthoughts

We have omitted some implementation details for the reader:

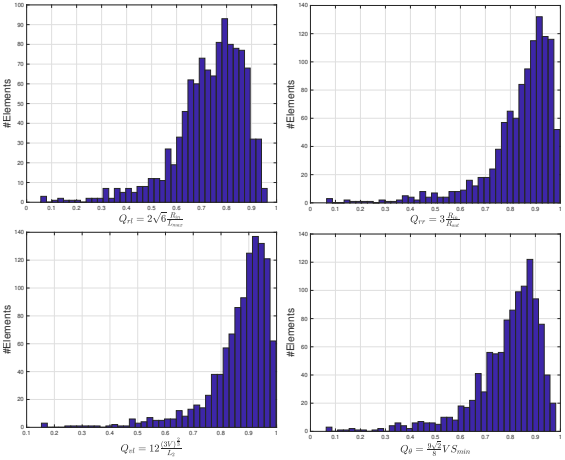
- How should one compute the coordinates `xA`, `yA`, `xB`, and `yB`?
- Is the resulting triangle mesh connected or not? Investigate how many times a non-grid vertex lying on the zero contour are added to the T-array.
- How can the implementation be changed to give a strong gurantee that there are no redundant vertices in the resulting triangle mesh?
- How can the quality of the resulting mesh be "easily" improved? (Hint: Consider the "Project Partcles" previously mentioned)

Further Reading

- J. R. Shewchuk: What Is a Good Linear Finite Element? Interpolation, Conditioning, Anisotropy, and Quality Measures, unpublished preprint, 2002.
- N. Molino, R. Bridson, J. Teran, and R. Fedkiw: A crystalline, red green strategy for meshing highly deformable objects with tetrahedra, Proc. International Meshing Roundtable 2003.
- P.-O. Persson, G. Strang, A Simple Mesh Generator in MATLAB. SIAM Review, Volume 46 (2), pp. 329-345, June 2004.
- J. Spillmann, M. Wagner, M. Teschner: Robust Tetrahedral Meshing of Triangle Soups, Proc. Vision, Modeling, Visualization. 2006
- M. K. Misztal, J. A. Bærentzen, F. Anton and K. Erleben, Tetrahedral Mesh Improvement Using Multi-face Retriangulation, 18th International Meshing Roundtable, 2009.

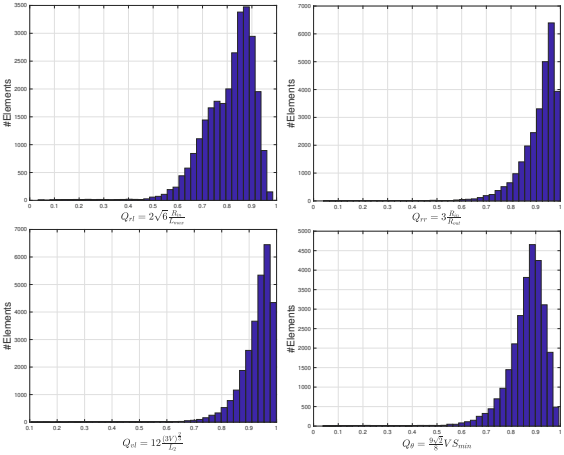
Example Quality Measure Plots

Bar with 1098 tetrahedra and 300 vertices



Example Quality Measure Plots

Refined Bar with 27977 tetrahedra and 5577 vertices



Assignment

- List different quality measures with pros and cons
- List different methods for creating tetrahedral meshes
- Discuss what is meant by a “good” mesh?
- Explain the DistMesh method to your fellow students
- Search the web for a paper about “marching tetrahedra” explain the algorithm to each other.

Assignment

- Find two or more quality measures from the Shewchuck paper that you believe will be good measures.
- Create 3-4 different tetrahedra or triangle meshes (use complex geometries – high curvature – non-convex) using different mesh generators
  - DistMesh
  - Matlab's **delaunay** function
  - Your own generator from your study group work
  - (If you up for a challenge try to include TetGen in your portfolio)
- For each of the meshes created compare histograms of quality measures and evaluate which method works best.