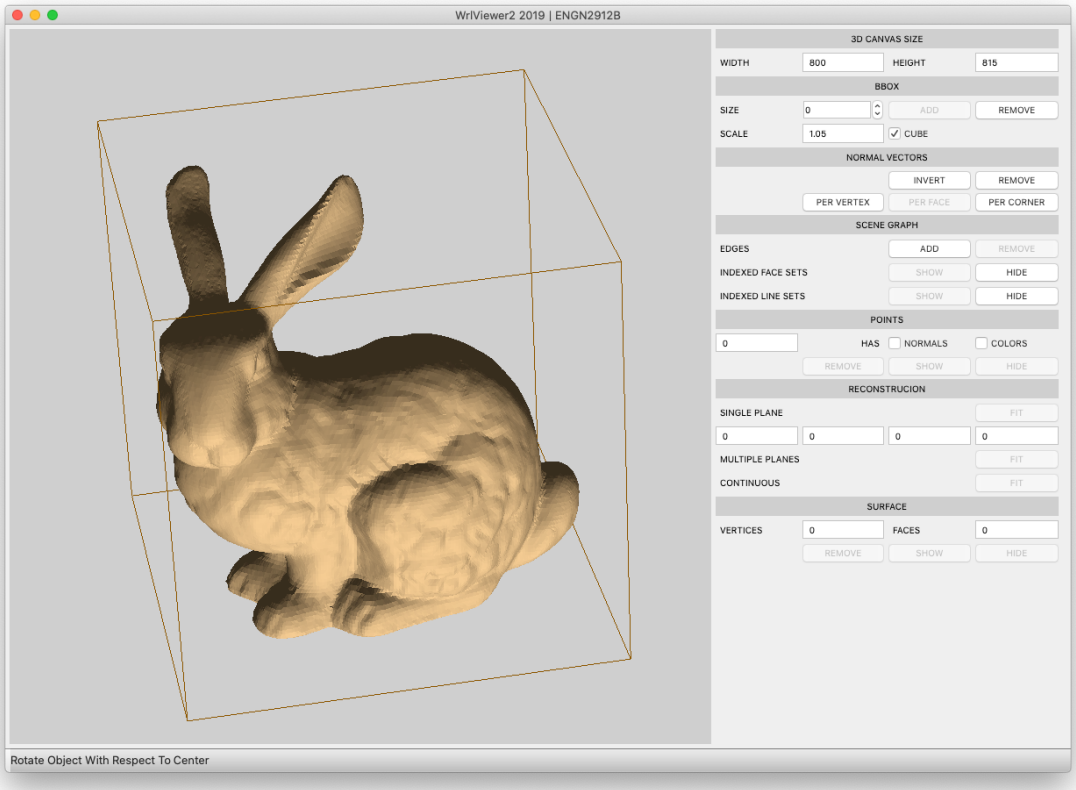
PDG2025-FCEN TP5 Notes



Disclaimer

These notes are based on the 2019 version of the software, which shows some differences in design and implementation. However the two versions share the same concepts. The current version and tasks are described in detail in the code README file, and source code comments.

SceneGraph Naming Conventions and Structure

All the surface reconstruction algorithms require a set of points and an hexahedral grid as input, and produce a polygon mesh as output. Each one of these three objects will be represented as a certain node node in the SceneGraph. For this assignment, and all the methods that you need to implement to complete it, we will make the following assumptions: 1) The bounding box is represented by a Shape node named “BOUNDING-BOX” containing an IndexedLineSet node as geometry. This Shape node should be a direct child of the top ScheneGraph node; 2) The set of input points is represented by a Shape node named “POINTS” containing an IndexedFaceSet node as geometry. This IndexedFaceSet node should have no faces, and as a result an empty coordIndex array. The only valid normal and color bindings for such an IndexedFaceSet are “NONE” and “PER_VERTEX”. We will further assume, and expect this IndexedFaceSet to have normals “PER_VERTEX”, which we will use in the surface reconstruction algorithms, but colors will be optional, since they will not be used in the surface reconstruction algorithms. The “POINTS” node should be a direct child of the top SceneGraph node; 3) The output surface is represented by a Shape node named “SURFACE” containing an IndexedFaceSet as geometry. The “SURFACE” node should also be a direct child of the top SceneGraph node. The

SceneGraph may also have other children, including Group or Transform nodes, which themselves may have children named “BOUNDING-BOX”, “POINTS”, or “SURFACE”, but those will not be the nodes that the user interface will operate on. Even though the VRML standard allows for Shape nodes to have NULL appearance fields, we will assume that every Shape node containing an IndexedFaceSet or IndexedLineSet as Geometry will contain an Appearance node in the appearance field, and this Appearance node will contain a Material node in the material field. Make sure when you create these nodes to also create add all these required nodes in the proper fields.

Bounding Box & Hexahedral Grid

The parameters which define the hexahedral grid are the SceneGraph bounding box `center` and `size` fields, as computed by the `updateBBox` method that you had to implement for the previous assignment, as well as by the `scale` parameter shown in the user interface, and the boolean variable `cube`, also shown in the user interface. The `scale` parameter should be a positive number, usually larger or equal than 1, used to scale the dimensions of the bounding box, as defined by the `size` variable. If the `cube` variable has the value `true`, then the maximum of the three `size` parameters is used as the dimension of all three bounding box sides. The `size` variable defined in the user interface is a positive integer `N` which is forced to be a power of 2 (`N=1<<depth`). It is used to subdivide each axis of the bounding box, resulting in a regular grid of N^3 cells. For each multi-index `(i,j,k)` with `0<=i,j,k<N` (or equivalently for every index `iCell` such that `0<=iCell<N*N*N`) there is a cell in the grid bound by the following `min` and `max` corners

```
min = (((N-i  )x0+(i  )x1)/N, ((N-j  )y0+(j  )y1)/N, ((N-k  )x0+(k  )z1)/N)
max = (((N-i-1)x0+(i+1)x1)/N, ((N-j-1)y0+(j+1)y1)/N, ((N-k-1)z0+(k+1)z1)/N)
```

where `(x0,y0,z0)` and `(x1,y1,z1)` are `min` and `max` corners of the overall bounding box, after the adjustments defined by the `scale` and `cube` variables. The variables `depth`, `cube`, and `scale` from the user interface, are stored as private variables in the class `gui/GuiViewerData`, which includes the following public access methods

```
int   getBBoxDepth();
void  setBBoxDepth(int d);
bool  getBBoxCube();
void  setBBoxCube(bool value);
float getBBoxScale();
void  setBBoxScale(float scale);
bool  hasBBox();
```

The following public methods are implemented in the SceneGraphProcessor class

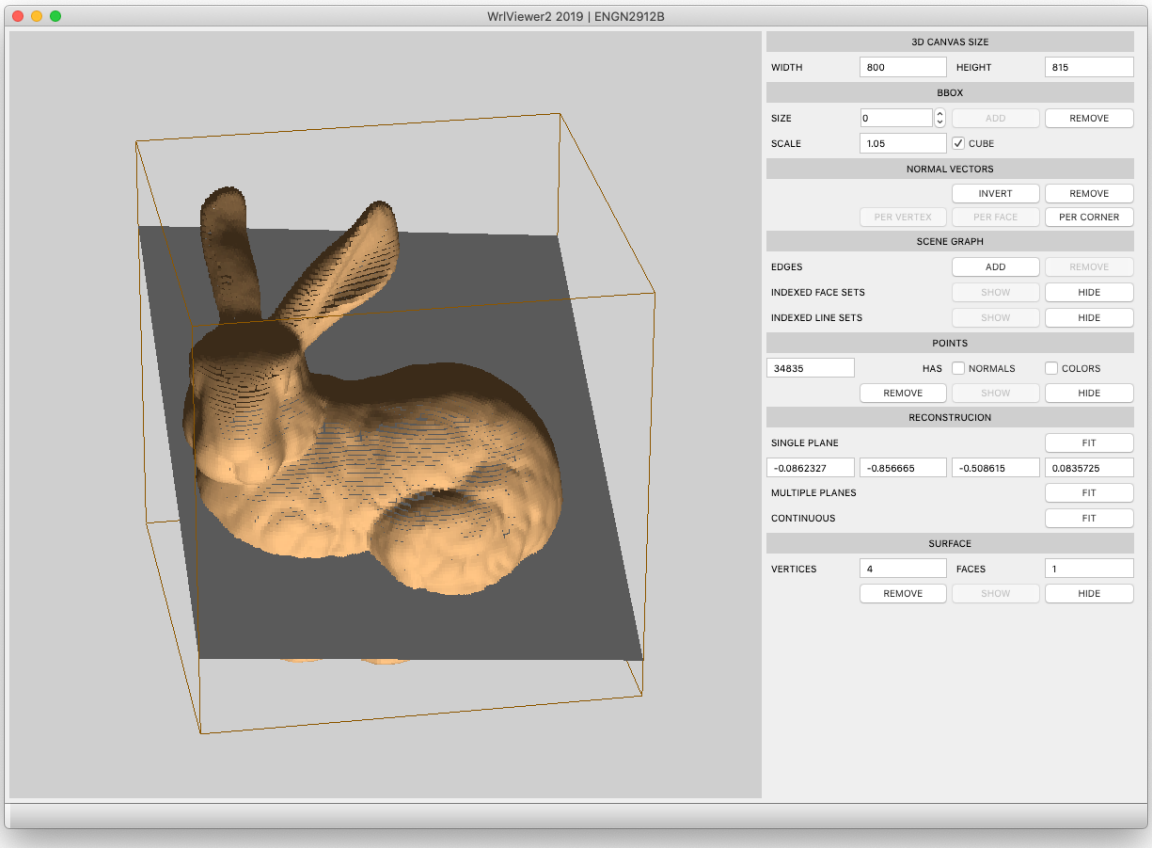
```
void gridAdd(int depth=0, float scale=1.0f, bool isCube=true);
void gridAdd(HexGridPartition& hgp);
void gridRemove();
bool hasGrid();
```

The `gridAdd` method first looks for a node named “GRID” amongst the children of the top SceneGraph. If the node exists, it should be a Shape node with an IndexedLineSet node as geometry. If these conditions are not satisfied, the node named “GRID” is removed from the scene graph and deleted, in which case the search should be performed again. If no valid node named “GRID” is found as a child of the SceneGraph, one is created and inserted. If a valid node named “GRID” is found as a child of the

SceneGraph, the IndexedLineSet is cleared. In either case, the result of this process is a Shape node named “GRID” which is a direct child of the top SceneGraph node, contains an empty IndexedLineSet as geometry, and an Appearance node as appearance. The Appearance contains a Material node as material. The `gridAdd` method computes the 3D coordinates of all the hexahedral grid vertices, ordered in lexicographical order, and stores them in the IndexedLineSet `coord` array. Then it determines which pairs of these vertex indices form edges in the hexahedral grid, and stores those edges as polylines of length 1 in the `coordIndex` array of the IndexedLineSet. An hexahedral grid of size N has (N+1)*(N+1)*(N+1) vertices and 3*N*(N+1)*(N+1) edges. The parameters defining the hexahedral mesh can be modified in the user interface while the scene graph does not yet have a node named “GRID”, in which case neither one of the previous two methods will be called. However, if the parameters defining the size or dimensions of the hexahedral grid are changed while the scene graph has a child named “GRID”, then the node will be deleted by calling the gridRemove method, and then recreated using the new parameters by calling the gridAdd method. Note that for large hexahedral grids, the “GRID” node may consume more memory than the set of points and the output polygon mesh combined, and as a result, the application may crash. For size N=128 and above, it is advisable to remove the “GRID” node from the scene graph before running any of the surface reconstruction algorithms.

Simple Surface Reconstruction Methods

Then we will start to implement implicit surface reconstruction methods. An implicit surface reconstruction method estimates a scalar function defined on a bounded domain, which contains a set of input points. In our case the domain of the scalar functions will be a box constructed as a function of the SceneGraph bounding box. A surface defined as a level set of the scalar function (typically level 0) approximates or interpolates the points. The polygon mesh is constructed as an approximation of the level set of the implicit function using an isosurface extraction algorithm.



We will work incrementally, first fitting a single plane to all the input points. The resulting surface will be a single polygon obtained by clipping the infinite fitting plane to the interior of the domain box. Then we will regularly subdivide the bounding box to create an hexahedral computation grid, and we will fit a plane within each cell of the grid which contains points. The output polygon mesh will be composed of all the polygons resulting from this computation To show the grid in the graphics window, you will need to extend the code that you wrote to generate an IndexedLineSet representation of the SceneGraph bounding box, to make it generate all the hexahedral grid vertices and edges of the hexahedral grid, while also taking into account a scale factor, and allowing for the bounding box to have equal sides or not. The resulting polygon mesh is composed of many disconnected polygons, because the scalar function being approximated by polygons in this case is a piecewise linear function, which is discontinuous across neighboring cells. We will make a third modification, which will make the polygonized surface continuous across neighboring cells. The polygon meshes resulting from this third algorithm still have a number of problems, which we will identify, but will not solve until next

assignment. As we develop the three algorithms described above, we will also incrementally construct a version of the isosurface construction algorithm known as Marching Cubes.

Fitting a Plane to a Set of Points as an Eigenvalue Problem

A plane can be represented as the set of zeros of a linear function $f(p) = n^t p + c$, where n is a nonzero 3D vector, p is a 3D point, and c is a scalar. If we restrict the vector n to be unit-length, then the value of the function measures the signed-distance from the point to the plane. A plane can be fit to a set of points $\{ p_1,\ldots p_N\}$ by minimizing the average of square distance from the points to the plane

$$E(n,c) = \frac{1}{N}\sum_{i=1}^{N}(n^t\, p_i + c)^2$$

under the constraint $n^2 = 1$. Since the parameter c is unconstrained, a necessary condition for minimum the minimum is that $c = -n^t\bar{p}$ where

$$\bar{p} = \frac{1}{N}\sum_{i=1}^{N} p_i$$

Replacing c in $E(n,c)$ we obtain

$$E(n) = \frac{1}{N}\sum_{i=1}^{N}(n^t\,(p_i - \bar{p}))^2 = n^t M\, n$$

where

$$M = \frac{1}{N}\sum_{i=1}^{N}(p_i - \bar{p})\,(p_i - \bar{p})^t$$

is a symmetric and non-negative definite 3x3 matrix. The solution to the problem of minimizing $E(n)$ under the constraint $n^2 = 1$ is the eigenvector of M associated with the minimum eigenvalue λ_1. We can use the following Eigen code fragment to solve the problem

```
Vector3d pMean; // compute from points
Matrix3d M;     // compute from points and pMean
SelfAdjointEigenSolver<Matrix3d> eigensolver(M);
Vector3d L(eigensolver.eigenvalues());
Matrix3d E(eigensolver.eigenvectors());
Vector3d n;
n << E(0,0), E(1,0), E(2,0);
Vector3d nMean; // compute from point normals
if(n.adjoint()*nMean<0) n = -n;
float c = -(n(0)*pMean(0)+ n(1)*pMean(1)+ n(2)*pMean(2));
```

Since the functions $f(p)$ and -$f(p)$ have the same zeros, if the points have associated normal vectors n_i, we pick n or $-n$ as the solution to the problem, depending on which one is closer to the average normal vector

$$\bar{n} = \frac{1}{N}\sum_{i=1}^{N} n_i$$

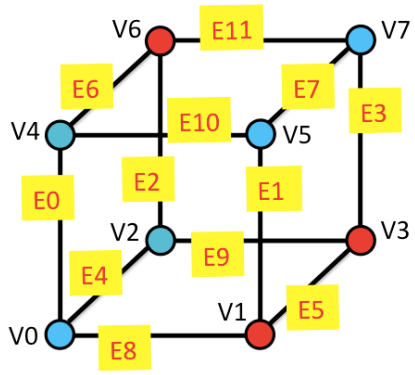
That is, if $n^t\bar{n} > 0$ we pick n; otherwise we pick $-n$.

Simpler Method to Fit a Plane to a Set of Oriented Points

When the points have associated normal vectors, an alternative is not to compute the minimum eigenvector, and pick $n^t = \bar{n}$, and keep $c = -n^t \bar{p}$ as before.

Marching Cubes

Marching cubes is a computer graphics algorithm, published in the 1987 SIGGRAPH proceedings by Lorensen and Cline for extracting a polygonal mesh of an isosurface from a three-dimensional scalar field (sometimes called voxels). An analogous two-dimensional method is called the marching squares algorithm. The algorithm proceeds through the scalar field, taking eight neighbor locations at a time (thus forming an imaginary cube), then determining the polygon(s) needed to represent the part of the isosurface that passes through this cube. The individual polygons are then fused into the desired surface. This is done by creating an index to a precalculated array of 256 possible polygon configurations (2^8=256) within the cube, by treating each of the 8 scalar values as a bit in an 8-bit integer. If the scalar's value is higher than the iso-value (i.e., it is inside the surface) then the appropriate bit is set to one, while if it is lower (outside), it is set to zero. The final value after all 8 scalars are checked, is the actual index to the polygon indices array. Finally each vertex of the generated polygons is placed on the appropriate position along the cube's edge by linearly interpolating the two scalar values that are connected by that edge.



In our case, we first evaluate the linear function on the eight corners of the bounding box, and we store the results in a float F[8] array. Then we determine on which corners the function is positive or negative, and we store the results in a bool B[8] array, where B[i]=(F[i]<0.0f). Then, for each of the 12 edges (i,j) of the cube, if B[i]!=B[j] an iso-vertex is supported on the edge. We compute the 3D coordinates of the iso-vertex by linearly interpolating the corresponding cube corner 3D coordinates, store the coordinates in the coord array of the output IndexedFaceSet, and the new vertex index in an cube-edge to iso-vertex index table int iE[12]. After all the iso-vertices have been computed and stored, we interconnect them forming a polygon face by looking at the configuration associated with the 8-bit integer defined by the bits B[0],…,B[7]. The method

static int makeCellFaces(bool B[/*8*/], int iE[/*12*/], vector<int>& coordIndex);

from the provided class util/IsoSurf can be used to perform the last step. The first argument is the array of sign bits. The second argument is the cube-edge to iso-vertex index table which we have to precompute. The class IsoSurf has the static private variable

```
static const int _edgeTable[12][2] = {
  { 0, 4 }, { 1, 5 }, { 2, 6 }, { 3, 7},
  { 0, 2 }, { 1, 3 }, { 4, 6 }, { 5, 7},
  { 0, 1 }, { 2, 3 }, { 4, 5 }, { 6, 7}
};
```

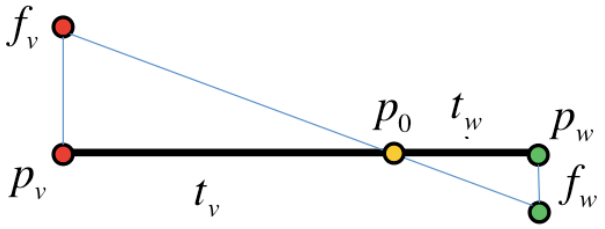
which can be accessed using the public method

```
static const Int2* getEdgeTable();
```

Each of the 12 entries in this table is a pair of cube corner indices representing a cube edge. The cube-edge to iso-vertex index table `iE[]` should be initialized with -1's . Some of those -1's will remain in the table after the iso-vertex computation to indicate that those edges do not support iso-surface vertices.

```
int i,j,k;
const Int2* edge = IsoSurf::getEdgeTable();
for(i=0;i<12;i++) {
  iViso = -1;
  j = edge[i][0];
  k = edge[i][1];
  if(B[j]!=B[k]) {
    iViso = (coordIso.size()/3);
    // get coordinates of grid vertices j and k
    // get function values of grid vertices j and k
    // compute coordinates of isovertex along edge
    // push_back onto coordIso
  }
  iE[i] = iViso;
}
```

To compute the coordinates of an iso-vertex Vertex3d v, along an edge `(i,j)` of the cube for which `B[i]!=B[j]` (or equivalently `F[i]*F[j]<0.0f`), with the cube corner vertices stored in the array `Vertex3d v[8]`, we have to find two values `float t0,t1` such that: 1) `0<=t0,t1<=1`; 2) `t0+t1=1`; 3) `v=t0*v[i]*t1*v[j];` and 4) `t0*F[i]+t1*F[j] = 0`. The solution is `t0=F[j]/(F[j]-F[i])` and `t1=F[i]/(F[i]-F[j]);`



$$p_0 = t_v p_v + t_w p_w$$

$$t_v = \frac{f_w}{f_w - f_v} \qquad t_w = \frac{f_v}{f_v - f_w}$$

Surface Reconstruction Methods to be implemented

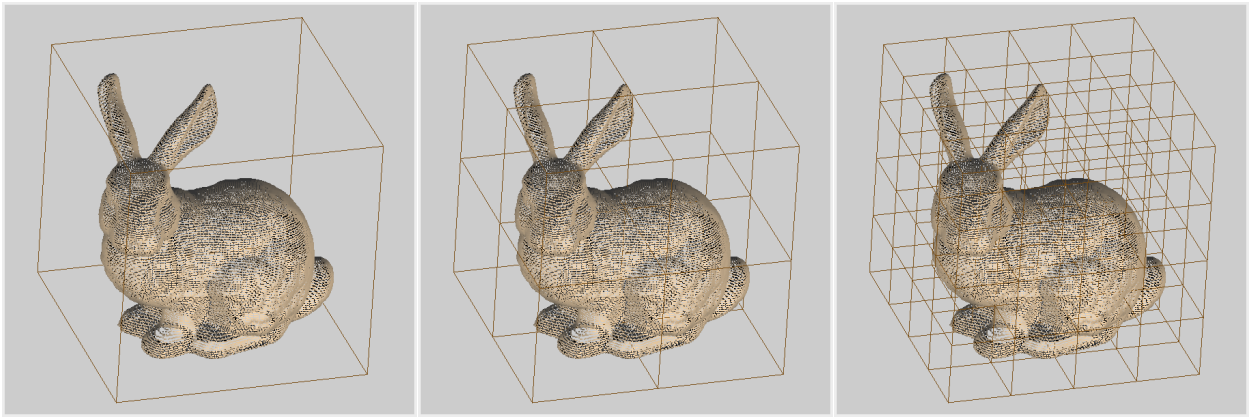
For this assignment you have to implement the following three methods of the class `wrl/SceneGraphProcessor`

```
  void fitSinglePlane
  (const Vec3f&   center,
   const Vec3f&   size,
   const float    scale,
   const bool     cube,
   Vec4f&         f);

  void fitMultiplePlanes
  (const Vec3f&   center,
   const Vec3f&   size,
   const int      depth,
   const float    scale,
   const bool     cube,
   vector<float>& fVec );

  void fitContinuous
  (const Vec3f&   center,
   const Vec3f&   size,
   const int      depth,
   const float    scale,
   const bool     cube);
```

Specifying the Bounding Box and the Hexahedral Grid



In the three methods the following variables specify the dimensions of the bounding box (depth is not passed as a parameter to the first method), and the subsequent subdivision into an hexahedral grid.

```
const Vec3f&   center;
const Vec3f&   size;
const int      depth;
const float    scale;
const bool     cube;
```

The variables center and size are obtained in the user interface from the scene graph, which you computed in a previous assignment, but you should not assume so. The two variables could have arbitrary values. If any element of the size variable is not positive, then the bounding box is empty, and the methods should return without performing any computation. The scale variable should be positive as well. The elements of the size variable should me multiplied by the scale variable to determine the preliminary dimensions of the bounding box. In general by setting the scale variable to a value larger than 1, we can make the bounding box larger to prevent artifacts associated by point being to close to the faces of the bounding box. But you should only assume that the variable scale is positive. If the variable scale is not positive, the three methods should return without performing any computation. If the boolean variable cube is true, all the edges of the bounding box should have the same length, equal to the maximum of the three components of the size variable, multiplied by the value of the scale variable. The depth variable is used to specify how to subdivide the bounding box into a regular hexahedral grid of N*N*N cells, where N=1<<depth. Note that this hexahedral grid has (N+1)*(N+1)*(N+1) vertices, which we can assign indices according to the lexicographic order

```
int i,j,k,iV;
for(i=0;i<=N;i++) {
  for(j=0;j<=N;j++) {
    for(k=0;k<=N;k++) {
      // grid vertex index
      iV = k+(N+1)*(j+(N+1)*i);
      // ...
    }
  }
}
```

void fitSinglePlane()

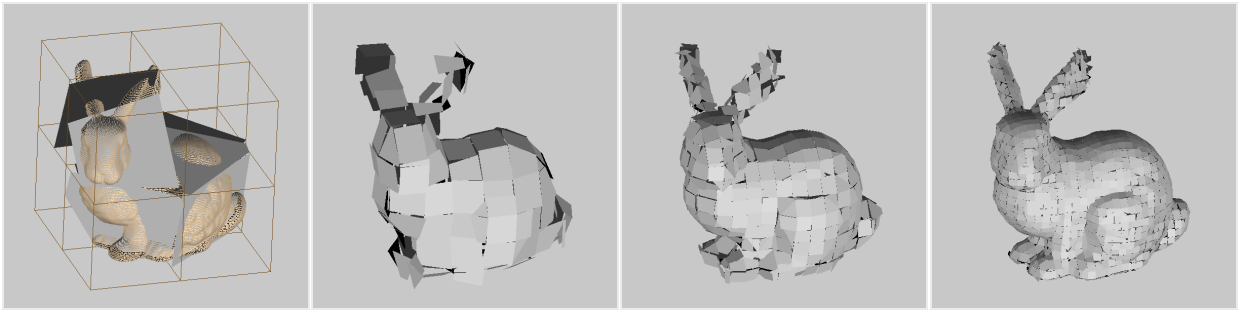
You have to implement this method primarily to debug the Marching Cubes algorithm within a single cube. You should implement this method in a way that would allow you to reuse the other methods. You can use the following skeleton to implement your code

```
void SceneGraphProcessor::fitSinglePlane
(const Vec3f& center, const Vec3f& size,
 const float scale, const bool cube, Vec4f& f) {

  // return if the bounding box is empty
  //
  // find the input point set in the scene graph
  // IndexedFaceSet* points  = (IndexedFaceSet*)0;
  // return if you do not find it
  // return if the IndexedFaceSet does not have normals per vertex
  // return if the number of points is too small
  //
  // compute the coordinates of the bounding box bounding corners
  // from the given parameters
  float x0=0,x1=0,y0=0,y1=0,z0=0,z1=0;
  // ...
  Vec3f min(x0,y0,z0);
  Vec3f max(x1,y1,z1);
  // compute the eight bbox corner coordinates
  // Vec3f v[8];
  // ...
  // fit a plane to the points contained in the bounding box
  //
  // find the output surface in the scene graph; if necessary
  // create a new instance and insert it in the scene graph
  IndexedFaceSet* plane = (IndexedFaceSet*)0;
  // ...
  // clear the output surface
  // ...
```

```
  // get the array of coordinates of the output surface
  vector<float>& coordIfs = plane->getCoord();
  // evaluate the linear function at the eight bounding box corners
  // also evaluate the function sign (1 if negative, 0 if non-negative)
  float F[8];
  // ...
  bool  b[8];
  // ...
  // get the edge table from the IsoSurf class
  const Int2* edge = IsoSurf::getEdgeTable();
  // compute the isovertex coordinates
  int iE[12],iV,i;
  for(i=0;i<12;i++) { // for each of the 12 edges of the cube
    //   set the isovertex index to 'no vertex' (-1)
    iV = -1;
    // if (sign at edge ends are different) {
    //   set the vertex index to the next available vertex index
    //   compute the vertex coordinates by linear interpolation
    //   save the vertex coordinates into the output coord array
    // }
    // save the vertex index in the edge-to-vertex array
    iE[i] = iV;
  }
  // create isosurface faces
  vector<int>& coordIndex = plane->getCoordIndex();
  // we should have coordIndex.size()==0 here
  // explain what this function does
  int nFaces = IsoSurf::makeCellFaces(b,iE,coordIndex);
  // save face normal
  plane->setNormalPerVertex(false);
  vector<float>& normal = plane->getNormal();
  // normal.size()==0 here
  normal.push_back(f.x);
  normal.push_back(f.y);
  normal.push_back(f.z);
}
```

void fitMultiplePlanes()



The next step is to subdivide the bounding box into an `N*N*N` hexahedral grid, and for each cell which contains points, fit a plane to those points, and add the polygon resulting from applying marching cubes to the resulting linear function evaluated at the vertices of the cell. The planes will be fitted to the points within each cell independently, and the vertices of the polygons generated within a cell will not be shared across cell boundaries.

The `minCell` and `maxCell` coordinates of a cell `(i,j,k)` with `0<=i,j,k<N`, can be computed from the indices `(i,j,k)` and the min and max coordinates of the bounding box:

```
minCell.x = ((N-i  )*min.x+(i  )*max.x)/(N);
maxCell.x = ((N-i-1)*min.x+(i+1)*max.x)/(N);
minCell.y = ((N-j  )*min.y+(j  )*max.y)/(N);
maxCell.y = ((N-j-1)*min.y+(j+1)*max.y)/(N);
minCell.z = ((N-k  )*min.z+(k  )*max.z)/(N);
maxCell.z = ((N-k-1)*min.z+(k+1)*max.z)/(N);
```

Since it would be extremely inefficient to test whether each point belongs to each cell, we will first generate a partition of the set of points. For each point with coordinates `(x,y,z)`, which cell `(i,j,k)` it belongs to can be determined in constant time by quantization

```
i = N*(x-min.x)/(max.x-min.x);
j = N*(y-min.y)/(max.y-min.y);
k = N*(z-min.z)/(max.z-min.z);
```

The partition can be represented as an array of linked-lists, with one linked list per cell. You can use the STL list class, or implement your own partition using two arrays of integers. The partition can be constructed by linear traversal of the set of points

```
int* next = new int[nPoints];
// initialize to -1's
int* first = new int[nCells];
// initialize to -1's
for(int iP=0;iP<nPoints;iP++) {
  // get the coordinates (x,y,z) of the point
  // determine which cell (i,j,k) the point belongs to
  // compute the lexicographic cell index
  int iCell = k+N*(j+N*i);
  // link the point iP to the iCell list
  next[iP] = first[iCell];
  first[iCell] = iP;
}
```

After the partition is constructed, whether a cell `iCell` contains points or not can be determined by looking at the value of `first[iCell]`. The cell contains points only if `first[iCell]>=0`. The list of points belonging to the cell `iCell` can be visited by a simple linear traversal of the corresponding list

```
for(int iP=first[iCell];iP>=0;iP=next[iP]) {
  // ...
}
```

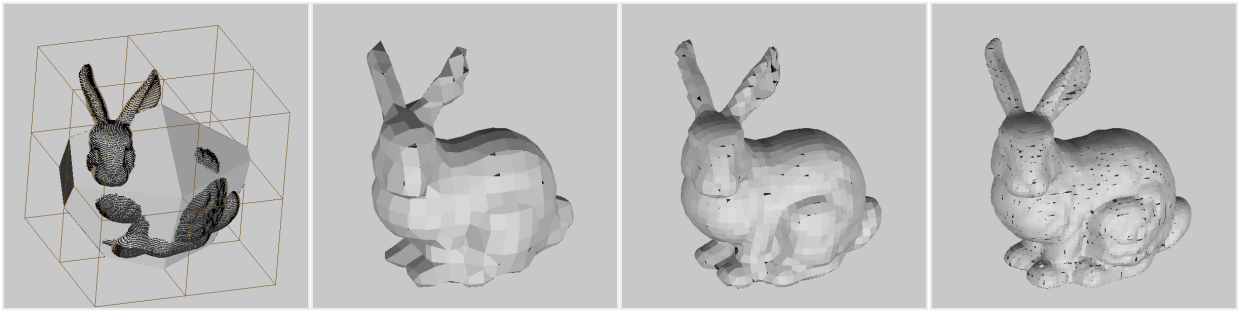
An implementation is already provided in the `SceneGraphProcessor` class

```
int SceneGraphProcessor::_createPartition
(Vec3f& min, Vec3f& max, int depth, vector<float>&coord);
```

Another implementation is also provided as the `HexGridPartition` class.

The overall structure of the `fitMultiplePlanes()` method should be very similar to the `fitSinglePlane()` method, although the output `IndexedFaceSet` node should be named "`SURFACE`" instead of "`PLANE`". You only have to add the construction of the partition, and a loop to visit all the cells. Within each cell that contains points, you should compute the `minCell` and `maxCell` coordinates, and then perform the same steps as in the `fitSinglePlane()` method.

void fitContinuous()



In the `fitMultiplePlanes()` vertices of polygons generated within one cell are not shared across cell boundaries, resulting in a discontinuous polygon mesh with so-called “cracks”. The first step towards solving this problem is to make sure that when the coordinates of a vertex are computed along an edge shared by multiple cells, the resulting values are the same independently of in which cell the computation is performed. Vertex indices of polygons generated within on cell will not yet be shared across cell boundaries, but if two cells share an edge, and that edge supports an isovertex, then we want to make sure that the same coordinates are obtained within each cell.

The reason for the “cracks” is that when we evaluate the linear functions at the vertices of the cell, we obtain different values. To solve this problem we need to compute common function values for the vertices of the hexahedral grid, so that the isosurface generation is performed as a function of these values. We can use a global array to store the function values, and we will have to visit the cells twice. In a first pass through the cells we can accumulate the global function values as the average of the values obtained by evaluating the linear functions estimated within each cell on the cell vertices. Since the number of cells contribute to each function value is variable, we need another variable for each grid vertex to count how many cells contributed to that particular function value.

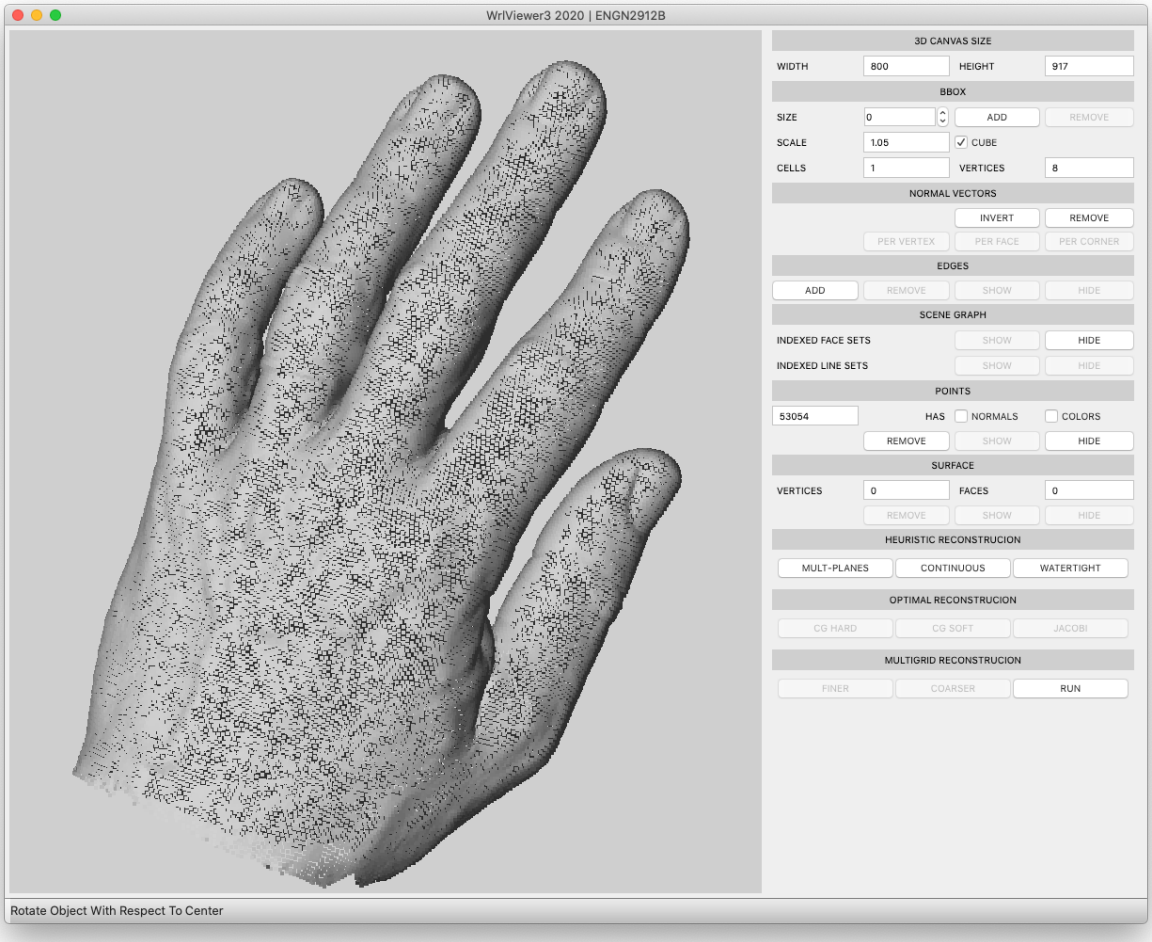
```
int nGridVertices = (N+1)*(N+1)*(N+1);
float* fGridVertex = new float[nGridVertices];
float* wGridVertex = new float[nGridVertices];
// initialize to 0’s
// for non empty each cell (i,j,k) {
//   fit linear function Vec4f f to the points contained in the cell
//   (make sure that linear function normal vector matches the average point normal)
//   for each corner of the cell {
//     determine the coordinates of the corner
//     evaluate the linear function
//     determine the grid vertex index iV of the corner
//     add the evaluated value to fGridVertex[iV]
//     increment wGridVertex[iV] by one
//   }
// }
// normalize function values
// for(iV=0;iV<nGridVertices;iV++)
//   if(wGridVertex[iV]>0.0f)
//     fGridVertex[iV] /= wGridVertex[iV];
```

Then we need a second pass through the non-empty cells to perform the isosurface generation as in the `fitMultiplePlanes()` method; except that now, rather than fitting the linear function to the points

and evaluating the function at the cell corners, we need to retrieve the function values for the corners from the `fGridVertex` table.

```
// for non empty each cell (i,j,k) {
//   for(i=0;i<8;i++) { // for each corner of the cell
//     determine the grid vertex index iV of the corner
//     get the function value fGridVertex[iV] and store it in F[i]
//     determine the function signs B[i] = (F[i]<0);
//   }
//   same as for fitSimplePlane() and fitMultiplePlanes()
// }
```

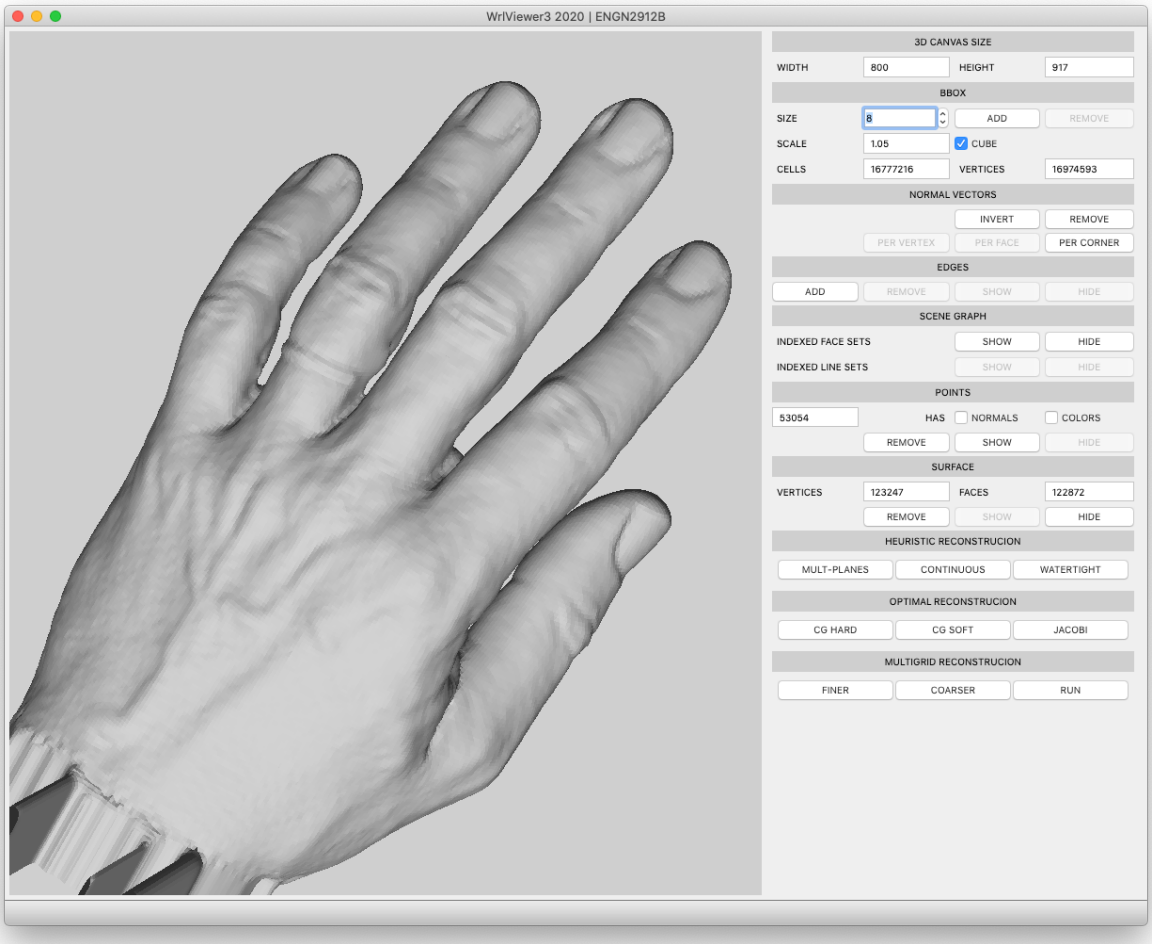
Additional Surface Reconstruction methods …
if we had more time or if you want to get extra credit



Here we would like to address and fix some problems with the method

```
void fitContinuous
(const Vec3f&   center,
 const Vec3f&   size,
 const int      depth,
 const float    scale,
 const bool     cube);
```

By storing common function values in an array associated with the vertices of the hexahedral grid, we were able to prevent geometric discontinuities, but we have two remaining problems to solve.



The SimpleGraphMap Class

The first remaining problem is that vertex indices of polygons generated within one cell are not shared across cell boundaries, even though when two cells share an edge, and that edge supports an isovertex, then the same coordinates are obtained within each cell, but up to four vertex indices may be generated for each isovertex. To solve this problem we need to use a data structure where the association between grid edges and isovertex indices is stored. The isovertex index should be allocated, and the coordinates computed, only the first time that a particular edge is visited while traversing the cells, but in subsequent visits to the same edge, the previously allocated vertex index should be retrieved and reused. You can implement this function for example using an STL map, which may or may not be very efficient. An alternative to solve this problem, is to implement our own class `util/SimpleGraphMap` with the following public interface

```
class SimpleGraphMap {
public:
  SimpleGraphMap();
  ~SimpleGraphMap();
  void   clear();
  // return (0<=jV && 0<=kV && jV!=kV && get(jV,kV)==-1)?value:-1
  int    insert(int jV, int kV, int value);
  // if (jV,kV) is found, return the value associated with (jV,kV)
```

```
  // otherwise it returns -1
  int    get(int jV, int kV) const;
};
```

If jV and kV are different grid vertex indices, and they form an edge in the hexahedral grid, the method

```
int get(int jV, int kV) const;
```

should return the isovertex index associated with that edge, if such an index and association was created before this function call; otherwise it should return -1 to indicate that no such isovertex exists yet, or that there is an error in the arguments. The method

```
int insert(int jV, int kV, int value);
```

should establish the association between the edge (jV,kV) and the third parameter value, if such an association can be established. If successful, it should return the value associated with the edge. It should return -1 if there is an error in the parameters, if the association cannot be established, or if the association already exists. After checking the validity of the parameters, insert() should first call the method get() to verify whether a value is already associated with the edge or not. Again, if the association exists, the method should fail returning the value -1.

The constructor should create an instance of the data structure with no association between grid edges and isovertex indices. Note that the dimensions of the hexahedral grid such as number of cells or vertices, are not passed as arguments to the constructor. Feel free to modify the constructor if you find a way to implement a more efficient version of this class by passing additional parameters.

```
void clear();
```

should reset the data structure to the state it was right after construction.

Since the the total number of edges of the hexahedral grid is 3*N*(N+1)*(N+1), and only a small number (O(N*N)) of all the edges will be associated with isovertex indices, using one or several three-dimensional arrays to implement this class is not a good idea, even though with such an implementation we would have constant time access to the associated values. At the expense of increasing the access time, we should try to use sparser data structures such as lists, lists of lists, or small arrays of lists. An hexahedral grid has three types of edges, aligned along the three principal axes. For a grid of N*N*N cells and (N+1)*(N+1)*(N+1) vertices, we can have edges of the form

(i,j,k) – (i,j,k+1)

where 0<=i,j<=N and 0<=k<N, of the form

(i,j,k) – (i,j+1,k)

where 0<=i,k<=N and 0<=j<N, of the form

(i,j,k) – (i+1,j,k)

where 0<=j,k<=N and 0<=i<N, of the form. We can implement this class using three arrays of dimension (N+1)*(N+1), and one variable length array of values

```
int* first0 = new int[(N+1)*(N+1)];
int* first1 = new int[(N+1)*(N+1)];
int* first2 = new int[(N+1)*(N+1)];
// initialize first0, first1, and first2 to -1's
vector<int> value;
```

The element `indx=first0[i*(N+1)+j]` would point into an element of the `value` array, which would describe the first element of a linked list where all the edges of the first type with the same `(i,j)` coordinates would be stored in arbitrary order. In fact, for edges of the first type the `value` array should contain a pair `(k,next)`, where `next` would be another index into the value array where the next element of the corresponding list is stored. In this case the insert() method could be implemented in this way

```
int insert(int jV, int kV) {
  // convert jV and kV to the form (i,j,k)
  // if jV is (i,j,k) and kV = (i,j,k+1) and get(jV,kV)==-1
  int next = first[i*(N+1)+j];
  first[i*(N+1)+j] = value.size();
  value.push_back(k);
  value.push_back(next);
  // if jV is (i,j,k) and kV = (i,j+1,k) and get(jV,kV)==-1
  // ...
  // if jV is (i,j,k) and kV = (i+1,j,k) and get(jV,kV)==-1
  // ...
}
```

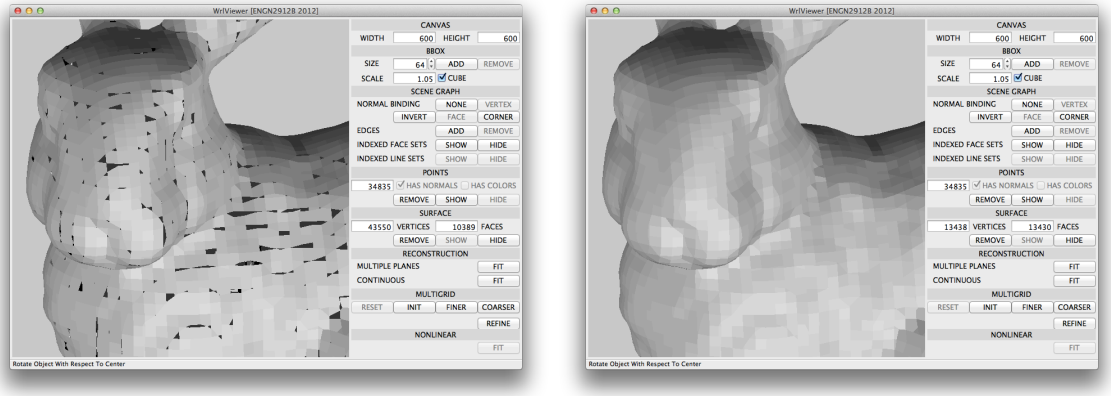
This is not the only way to implement this class, but the public interface to the class isolates the specific implementation from your use of it. You should experiment with different ideas until you get a satisfactory implementation.

Sharing Isovertex Indices Across Cells

The first loop of `fitContinuous()`, where the non-empty cells are visited, the local linear function are fit, and the shared function values are computed for the grid vertices does not require changes. The second loop, where the non-empty cells are visited a second time and the isosurface is generated requires minor changes

```
int i,j,k,iV,iVB[8],iE[12];
float F[8];
bool  B[8];
Vec3f v[8];
const Int2* edge = IsoSurf::getEdgeTable();
vector<float>  coordIsosurf;
vector<int>    coordIndexIsosurf;
SimpleGraphMap map;
for (each non-empty cell) {
  for(i=0;i<8;i++) { // for each cell vertex
  // get grid vertex coordinates v[i]
  // get grid vertex index iVB[i]
  // get function value and sign B[i] = ((F[i]=fGrid[iVB[i]])<0.0f);
  for(i=0;i<12;i++) { // for each cell edge
    iV   = -1;
    j    = edge[i][0];
    k    = edge[i][1];
    if(B[j]!=B[k]) {
    // look for associated isovertex index
    iV = map.get(iVB[j],iVB[k]);
    if(iV<0) { // need to create a new vertex
      // assign new isovertex index
      iV = (int)((coordIsosurf.size()/3));
      // compute the isovertex coordinates and push_back() onto coordIsosurf;
      // save the edge->isovertex association
      map.insert(iVB[j],iVB[k],iV);
    }
    iE[i] = iV;
  }
  // interconnect isovertices to form isofaces
  IsoSurf::makeCellFaces(b,iE,coordIndexIsosurf);
}
```

Filling Holes



The only remaining problem with the fitContinuous method is that the resulting polygon mesh seems to be missing polygons, as we can see in the figures. This problem is a consequence of the fact that we do not generate isosurface faces within cells not containing points. We do so because so far we are not assigning function values to all the grid vertices, but just to those which are corners of occupied cells. If every grid vertex had a meaningful function value, then we could run marching cubes on all the cells, whether they contain points or not, an produce a so-called “watertight” polygon mesh, such as the one shown in the figure above. We will first implement a heuristic method named “propagating wavefront” to incrementally assign function values to those vertices which do not have valid values. Then we will discuss and implement methods based on global optimization techniques, which will use the values produced by the propagating wavefront method as initial estimates.

The method to extend the definition of function values from those which are corners of occupied cells to all the grid vertices, should be applied in between the two loops of the method `fitContinuous()`, after the loop which visits the occupied cells and accumulates the function values and the weights in the arrays

```
int* fGridVertex = new float[nGridVertices];
int* wGridVertex = new float[nGridVertices];
```

and before the loop which performs the isosurface construction. In fact, since the second loop now needs to visit all the cells, and no longer needs to check for occupied cells, we can split the method into two separate methods; one to generate the function and weight values, where `fGrid` is here an input parameter which replaces the `fGridVertex` array, and another to generate the isosurface from the grid and function values

```
void fitWatertight
  (const Vec3f&   center,
   const Vec3f&   size,
   const int      depth,
   const float    scale,
   const bool     cube,
   vector<float>& fGrid /* output */);

void computeIsosurface
  (const Vec3f&   center,
   const Vec3f&   size,
   const int      depth,
```

```
   const float    scale,
   const bool     cube,
   vector<float>& fGrid /* input */);
```

In addition to the first loop of the `fitContinuous()` method, this `fitWatertight` () method should also take care of extending the function values to all the grid vertices, and finally to call the `computeIsosurface` () method to generate a new polygon mesh.

Defining Function Values by Wavefront Propagation

After the accumulation loop, we normalize the function values for those vertices which are corners of occupied cells. In the previous methods the array wGridVertex is no longer used after this point. However, since the positive values of wGridVertex correspond to vertices where the function is defined, we can use it to implement our wavefront propagation algorithm. The idea is to first propagate the function from the grid vertices where it is defined to the first order neighbors, and then iterate until all the function is defined on all the vertices. For this we need to identify all the grid edges where the function is defined on one end and not defined in the other. The defined function value is accumulated for the undefined vertex. Since several vertices with defined function value may contribute to the function value of an undefined vertex, we need an accumulator to count how many contributions we have collected for each vertex. We can use the fGridVertex to accumulate the values, and the wGridVertex to count the number of contributions, since the values stored in wGridVertex for the vertices where the function was not defined is zero. However, as soon as values start to accumulate, the wGridVertex values no longer determine which group each vertex belongs to. We can solve this by setting the wGridVertex values to a negative value when we normalize the function values

```
for(iV=0;iV<nGridVertices;iV++) {
  if(wGrid[iV]>0.0f) {
    fGrid[iV] /= wGrid[iV];
    wGrid[iV] = -2.0f; // to indicate original vertices
  }
}
```

By doing this, vertices with negative wGridVertex value are those where the function is already defined, vertices with positive wGridVertex value are those on the wavefront where the function values are being accumulated, and vertices with zero wGridVertex value are those not yet reached by the wavefront. The overall wavefront propagation algorithm can be structured as follows

```
int newValues = 0;
do {
  for each grid edge (jV,kV) {
    if function is defined on jV but not on kV {
      fGridVertex[kV] += fGridVertex[jV];
      wGridVertex[kV] += 1.0f;
    } else if function is defined on kV but not on jV {
      fGridVertex[jV] += fGridVertex[kV];
      wGridVertex[jV] += 1.0f;
    }
  }
  newValues = 0;
  for each vertex iV such that wGridVertex[iV]>0.0f {
    newValues++;
```

```
    fGrid[iV] /= wGrid[iV];
    wGrid[iV] = -2.0f; // to indicate vertex with defined function value
  }
} while (newValues>0);
```

By counting how many new function values have been defined we can determine when to stop. The first loop of the wavefront propagation algorithm can be implemented with a loop over all the edges of the grid, but that would be very inefficient for large grids. Instead, a list of grid vertices which are at the border of the wavefront can be generated within the initial normalization loop

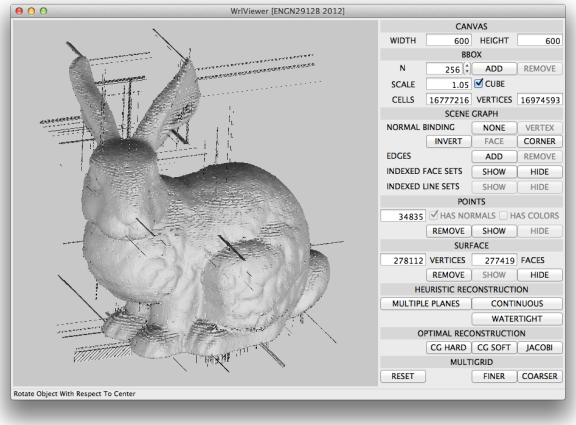
```
vector<int> source;
for(iV=0;iV<nGridVertices;iV++) {
  if(wGrid[iV]>0.0f) {
    fGrid[iV] /= wGrid[iV];
    wGrid[iV] = -2.0f; // to indicate original vertices
    source.push_back(iV);
  }
}
```

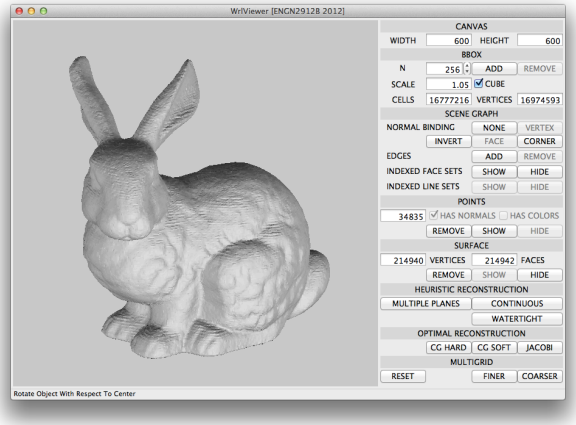
and then only the six edges incident to each one of these vertices can be checked. The list can be rebuilt when the inner normalization loop is run. The `newValues` variable is no longer needed, since it is equal to `source.size()`. A more efficient version of the wavefront propagation algorithm could be structured as follows

```
while(source.size()>0) { // while the wavefront is not empty
  for each vertex jV in source { // jV->(i,j,k)
    // by construction function is defined on jV
    for each first order neighbor kV of jV {
      // there are up to 6 of these
      // kV -> (i,j,k-1),(i,j,k+1),(i,j-1,k),(i,j+1,k), (i-1,j,k),(i+1,j,k)
      if(wGridVertex[kV]>=0.0f) { // function is not defined on kV
        // if first visit to kV, we should save kV in another list
        fGridVertex[kV] += fGridVertex[jV];
        wGridVertex[kV] += 1.0f;
    }
  }
  source.clear(); // clear the wavefront sources
  for each vertex kV such that wGridVertex[kV]>0.0f {
    // these are the vertices kV saved in the second list
    // when they were visited for the first time
    fGrid[kV] /= wGrid[kV];
    wGrid[kV] = -1.0f; // to indicate new vertex with defined function value
    source.push_back(kV); // add vertex to the list of new wavefront sources
  }
}
```

By propagating the function values in this way, this process tends to make the function values approximately constant far away from the points, which is not necessarily the best strategy since it may result in the “wrong” function value signs assigned to certain vertices, and these wrong signs result in visible artifacts in the isosurfaces, as we can see in the following picture. The picture on the right shows the result obtained using an iterative optimization-based approach to refine the implicit function estimate produced by the wavefront propagation algorithm. The wavefront propagation algorithm is based on heuristics, but it provides a good initial estimate which speeds-up the convergence of the

optimization-based iterative algorithm quite significantly. Our next task is to learn about and implement some some of these optimization-based algorithms.





Implicit Surface Reconstruction Methods Based on Optimization

An alternative strategy to extend the function values from the vertices of occupied cells to all the vertices of the grid is to formulate the problem as an optimization problem. There are many possible formulations based on different energy functions, as well as many different ways of implementing these algorithms. We will continue with our incremental strategy. We will start with a simple approach, we will implement it, and then we will make some changes to make it better.

As a first energy function to minimize, we can consider the sum, over all the grid edges, of the squares of the differences of the function values at the edge ends

$$\sum_{(i,j)} (f_i - f_j)^2$$

We will regard this energy function as a function of only the vertices which are not corners of occupied cells, and we will regard the values of the function at the corners of the occupied cells as constant. The values of the function at the corners of the occupied cells, which we will compute as in the previous methods, will be considered as "hard" constraints in this optimization problem. Minimizing this energy function while satisfying the constraints tends to make the implicit function as constant as possible far away from the data points.

We collect the the implicit function values associated with the grid vertices which are not corners of occupied cells as a vector of free variables F of dimension n. The quadratic energy function can be written in standard form as follows

$$\sum_{(i,j)} (f_i - f_j)^2 = F^t AF - 2b^t F + c$$

where the nxn matrix A is symmetric and non-negative definite, b is a vector of the same dimension as F, and c is a constant. In almost all practical problems the matrix A is positive definite. The values of the function at the corners of the occupied cells determine the values of the vector b and the constant c. The set of edges incident to the vertices which are not corners of occupied cells form a graph G, and the matrix A is only function of the combinatorial structure of the graph G. In fact, A is the so-called Laplacian matrix of the graph G. The matrix A and the vector b can be filled during a linear traversal of the grid edges. For each grid edge (i,j), if both vertex indices are free, then the term $(f_i - f_j)^2$ contributes to four components of the matrix A. If one vertex index is free and the other is constrained, then the term $(f_i - f_j)^2$ contributes to only one component of the matrix A, and also to a component of the vector b. If neither one of the two vertices is free, then the term contributes to the constant c, but since the linear system is independent of c, there is no need t accumulate it.

A necessary and sufficient condition for the minimum of a quadratic energy function like this one is that the gradient of the energy function be equal to zero. Equivalently, the minimizer of our problem if the solution of the linear system

$$AF = b$$

To solve our surface reconstruction problem we need to solve this linear system, and then transfer the values stored in the vector F to the corresponding components of the array `fGrid` which we will then

use to recomputed the isosurface. The problems are practical and related to the implementation. You will have to implement the following method as a member of the `SceneGraphProcessor` class

```
void optimalCGHard
  (const Vec3f&   center,
   const Vec3f&   size,
   const int      depth,
   const float    scale,
   const bool     cube,
   vector<float>& fGrid);
```

where the linear system will be solve using the Conjugate Gradients algorithm as implemented in the Eigen package. The method should start by determining the dimensions of the bounding box, creating the partitioning of the points associated with the grid, fitting the local linear function within each occupied cell, and computing the averages of these linear functions on the vertices of the occupied cells. You should reuse the code you wrote as part of the `fitContinuous()` and `fitWatertight()` methods. Then you need to establish the correspondence between grid vertex indices and free variables. The list of vertices of occupied cells can be collected as part of the traversal used to fit the local linear functions to the cells. The list of vertices which are not vertices of occupied cells are the remaining grid vertices. This list can be represented as an `vector<int> iVfree` array. The two lists can also be made by traversing the `wGrid` array after the evaluation of the implicit function at the corners of occupied cells. You may also need to construct additional arrays to map from grid vertex indices onto free vertex indices (indices into the vector of free variables F).

```
#include <Eigen/Dense>
#include <Eigen/Sparse>

void optimalCGHard (..., vector<float>& fGrid) {
  // ...
  // list of free grid vertex indices
  vector<int> iVfree;
  // fill during first traversal used to compute constrained function values
  // store the computed constrained function values in the fGrid array
  // but do not modify the fGrid values corresponding to free variable
  // since they will be used by the solver as initial estimates
  int n = nFreeGridVertices; // computed during previous traversal
  // allocate the arrays needed to solve the linear system
  VectorXd F(n),B(n); // these vectors are not initialized
  B = VectorXd::Zero(n); // initialize to zero
  // allocate the matrix needed to solve the linear system
  SparseMatrix<double> A(n,n); // this matrix is implicitly initialized to all 0's
  A.reserve(VectorXi::Constant(n,8)); // each grid vertex may have up to 8 neighbors
  // create and initialize a Conjugate Gradient solver for the matrix A
  ConjugateGradient<SparseMatrix<double> > solver;
  solver.compute(A);
  // fill the matrix A and the vector B
  for(each grid edge (jV,kV)) {
    // contributions to A and B by the term (f_jV-f_kV)^2
    if(both jV and kV are free vertices) {
      // let j so that jV==iVFree[j];
      // let k so that kV==iVFree[k];
      A.coeffRef(j,j) += 1.0; A.coeffRef(j,k) -= 1.0;
      A.coeffRef(k,j) -= 1.0; A.coeffRef(k,k) += 1.0;
    } else if(jV is a free vertex) {
      // let j so that jV==iVFree[j];
      A.coeffRef(j,j) += 1.0; B(j) += fGrid[kV];
    } else if(kV is a free vertex) {
      // let k so that kV==iVFree[k];
      A.coeffRef(k,k) += 1.0; B(k) += fGrid[jV];
    } // nothing to accumulate if both jV and kV are constrained
  }
  // fill F with initial values passed in fGrid as free variables
  for(i=0;i<iVFree.size();i++)
    F(i) = fGrid[iVfree[i]];
  // solve using given initial estimate
  F = solver.solveWithGuess(B,F);
  // alternatively, solve starting from F=0
  // F = solver.solve(B);
  // print some useful information provided by the solver
  cerr << "    " << solver.iterations() << " iterations"<< endl;
  cerr << "    " << solver.error() << " estimated error"<< endl;
  // transfer the resulting free variable values back to the fGrid array
  for(i=0;i<iVFree.size();i++)
    fGrid[iVFree[i]] = F(i);
  // recomputed isosurface
  // ...
}
```

Optimization-Based Implicit Surface Reconstruction With Soft Constraints

An alternative to performing the optimization with soft constraints is to include the constraints as an additional term in the energy function. We now consider the following energy function

$$\sum_i (f_i - f_i^0)^2 + \lambda \sum_{(i,j)} (f_i - f_j)^2$$

where the first sum is over all constrained vertices, f_i^0 is the target value assigned the constrained vertex computed as the average of the local linear functions evaluated at that vertex, the second sum is over all the grid edges, and λ is a positive scale parameter. The implementation is somehow simplified because now all the variables are free. As a result, we do not need to map back and forth between grid vertex indices and free vertex indices. The wGrid array can be used to determine which vertices are constrained. This information is needed to fill the matrix A and the vector b.

```
int n = nGridVertices;
// Allocate F,B, and A as in the previous case
// scale parameter; should be passed as parameter
double lambda = 0.05;
// fill A and b
for(iV=0;iV<n;iV++) {
  if(wGrid[iV]>0.0f) {
    A.coeffRef(iV,iV) += 1.0; B(iV) = fGrid[iV];
  } else {
    B(iV) = 0.0f;
  }
}
for(each grid edge (jV,kV)) {
  A.coeffRef(jV,jV) += lambda; A.coeffRef(jV,kV) -= lambda;
  A.coeffRef(kV,jV) -= lambda; A.coeffRef(kV,kV) += lambda;
}
```

You will have to implement this strategy as the following method as a member of the `SceneGraphProcessor` class

```
void optimalCGSoft
  (const Vec3f&   center,
   const Vec3f&   size,
   const int      depth,
   const float    scale,
   const bool     cube,
   vector<float>& fGrid /* input & output */);
```

The Jacobi Iteration

Once you implement the previous two methods and test them you will notice that the results are better than those produced by the heuristic approaches which we implemented before, but the performance is not very good. This is mainly due to two reasons. First, the number of free variables grow by a factor of almost 10 from one grid resolution to the next. This problem is normally addressed and solved replacing the regular grid by an adaptive grid, such as an octree. We will not attempt to do so in this course. The second problem is the conversion of data structures needed to call the packaged linear solver. We have our data stored in a particular form, but the solver needs it in a different form. As a result, new data structures need to be created and filled (matrix A, vector b, solution vector F, mappings from grid vertices onto solution indices, etc.), the solver is called, and then the solution needs to be transferred back to our original data structures. In cases like these, it is often much more efficient to implement your own iterative algorithm on top of your own data structures. And it is often the case that the simplest algorithm works well in practice. Perhaps the simplest iterative linear solver for a positive definite diagonal dominated linear system like ours is the Jacobi method. To solve the linear system $AF = b$, let D be the diagonal of the matrix A represented as a matrix of the same dimensions as A, and let $dF = D^{-1}(b - AF)$, which is a descent direction for the quadratic energy $E(F) = F^t AF - 2b^t F$. The Jacobi update is $F \rightarrow F + \varepsilon dF$ where ε is a small number. The optimal value can be computed in closed form as $\varepsilon = (dF^t(b - AF))/(dF^t AdF)$, but $\varepsilon = 1/2$ works well in practice. The Jacobi update should then be repeated until convergence, or for a fixed number of steps. The most interesting aspect of this algorithm is that neither the matrix A nor the vector b need to be represented explicitly. The descent vector dF can be accumulated during a traversal of the existing data structures. For the energy function with soft constraints

$$\sum_i (f_i - f_i^0)^2 + \lambda \sum_{(i,j)} (f_i - f_j)^2$$

the descent vector is computed by first traversing the constrained vertices, and then the grid edges. This code segment illustrates how to implement the Jacobi solver for our problem

```
vector<float> f;
// copy fGrid onto f
vector<float> df;
vector<float> wf;
while (stop criterion not met) {
  // zero accumulators
  df.clear();
  df.insert(df.end(),n,0.0f);
  wf.clear();
  wf.insert(wf.end(),n,0.0f);
  // accumulate soft constraints
  for(each constrained grid vertex iV) {
    df[iV] += (fGrid[iV]-f[iV]);
    wf[iV] += 1.0f;
  }
  // accumulate Laplacian
  for(each grid edge (jV,kV)) {
    df[jV] += lambda*(f[kV]-f[jV]);
    wf[jV] += lambda;
    df[kV] += lambda*(f[jV]-f[kV]);
    wf[kV] += lambda;
  }
```

```
  // normalize
  for(iV=0;iV<n;iV++)
    df[iV] /= wf[iV];
  // update
  for(iV=0;iV<n;iV++)
    f[iV] += mu*df[iV];
}
// copy f back onto fGrid
```

You will have to implement this strategy as the following method as a member of the `SceneGraphProcessor` class

```
void optimalJacobi
  (const Vec3f&   center,
   const Vec3f&   size,
   const int      depth,
   const float    scale,
   const bool     cube,
   vector<float>& fGrid /* input & output */);
```

Cascading Multigrid

A complementary strategy to speed up convergence is cascading multigrid. We have chosen to restricts our grids to those which double in size at each step to be able to implement cascading multigrid. The idea here is to solve the problem at a certain resolution, refine the grid and transfer the function values from the coarse to the fine grid by interpolation, start the iterative solver at the finer grid from the interpolated values, and the repeat the process until you reach the desired resolution. To interpolate from one grid resolution to the next, the coarse vertex values are first copied onto the fine vertex values corresponding to even vertex indices. The remaining fine vertices are supported on coarse edges, faces, or cells. In each of those cases the function value is computed as the average of the coarse function values assigned to the corners of the edge, face, or cell. The inverse operation, to transfer an implicit function from a fine grid to a coarse grid, is a lot simpler, since it only requires subsampling.

To complete this assignment you will have to implement this strategy as the following two methods of the `SceneGraphProcessor` class

```
void multiGridFiner
  (const Vec3f&   center,
   const Vec3f&   size,
   const int      depth,
   const float    scale,
   const bool     cube,
   vector<float>& fGrid /* input & output */);

void multiGridCoarser
  (const Vec3f&   center,
   const Vec3f&   size,
   const int      depth,
   const float    scale,
   const bool     cube,
   vector<float>& fGrid /* input & output */);
```