Point Set Processing for VTK - Outlier Removal, Curvature Estimation, Normal Estimation, Normal Orientation

Release 0.00

David Doria

December 4, 2009

Rensselaer Polytechnic Institute, Troy NY

Abstract

This document presents a set of classes (*vtkPointSetOutlierRemoval*, *vtkPointSetNormalEstimation*, *vtkPointSetNormalOrientation*, *vtkPointSetCurvatureEstimation*, *vtkEuclideanMinimumSpanningTree*, and *vtkRiemannianGraphFilter*) to enable several basic operations on point sets. These classes are implemented as VTK filters. Paraview plugin interfaces to the filters are also provided to allow extremely easy experimentation with the new functionality. We propose these classes as an addition to the Visualization Toolkit.

Contents

1 Introduction **2**

2 Outlier Removal - vtkPointSetOutlierRemoval **2**
- 2.1 Demonstration . . . 3
- 2.2 Code Snippet . . . 3

3 Normal Estimation - vtkPointSetNormalEstimation **3**
- 3.1 Demonstration . . . 4
- 3.2 Code Snippet . . . 4

4 Normal Orientation - vtkPointSetNormalOrientation **4**
- 4.1 Creating a Euclidean Minimum Spanning Tree . . . 5
  - Code Snippet . . . 5
- 4.2 Creating a Riemannian Graph on a Point Set - vtkRiemannianGraphFilter . . . 5
  - Code Snippet . . . 5

4.3 Orientation/Propagation Algorithm 6
Initialization 6
Propagation 6
4.4 Demonstration 6
4.5 Code Snippet 6

5 Curvature Estimation - vtkPointSetCurvatureEstimation **7**
5.1 Demonstration 7
5.2 Code Snippet 7

6 Future Work **8**

1 Introduction

In the last several years, an increasing number of tools produce 3D points as output. Examples include Light Detection and Ranging (LiDAR) scanners, Structure From Motion (SFM) algorithms, and Multi View Stereo (MVS) algorithms. These unordered point sets (or point "clouds") are typically provided simply as a list of 3D coordinates. There are many factors in all of these processes that lead to many of the points that are provided being "outliers". That is, several points do not seem to actually come from the surface that we expect. This will severely corrupt the results of many algorithms on this type of data. To remove outliers, we provide *vtkOutlierRemoval*.

By definition, point sets do not contain any connectivity information. This makes it impossible to apply many algorithms for 3D data processing. At the very least, point normals at each point are required. That is, if there was a surface through the points, what would the normal of the surface be evaluated at the points in the point set? The *vtkPointSetNormalEstimation* class performs this estimation. The algorithm we use to compute these normals has no concept of "inside" and "outside" of the object, so the orientation of the normals from point to point my not be consistent. As many algorithms rely on this orientation, we must attempt to correct the normals so they are consistently oriented. This is the role of *vtkPointSetNormalOrientation*. Necessary for the orientation algorithm are two graph algorithm implementations, *vtkEuclideanMinimumSpanningTree* and *vtkRiemannianGraphFilter*.

An estimate of the curvature of a point set is often a valuable tool. While the exact values of well defined mathematic quantities can be computed on a mesh, since we do not have connectivity information in a point set, an estimate will have to suffice. We provide *vtkPointSetCurvatureEstimation* to compute a heuristic idea of curvature at each point.

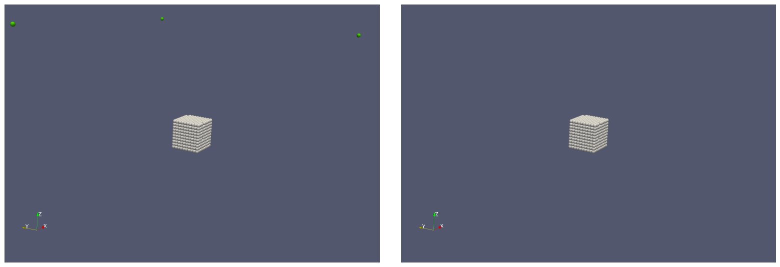
This set of classes provides these basic functionalities as well as a basis for further point set and surface processing algorithms for VTK.

2 Outlier Removal - vtkPointSetOutlierRemoval

We take the simple definition of an outlier to be a point that is farther away from its nearest neighbor than expected. To implement this definition, for every point p in the point set, we compute the distance from p to the nearest point to p. We sort these distances and keep points whose nearest point is in a certain percentile of the entire point set. This parameter is specified by the user as *PercentToRemove*.

2.1 Demonstration

To demonstrate outlier removal, we have created a cube of points, shown in Figure 1(a). We have added to this cube three spurious points (shown in green near the top of the figure). In Figure 1(b), we show the resulting point set after 1% of points have been removed.



(a) Input Pointset (b) Outliers Removed

Figure 1: Outlier removal demonstration.

In the current implementation, exactly the percentage of the points specified are removed. In future work, we plan to add a "do not remove" threshold which will not remove points which are "definitely not outliers" according to a specified criterion.

2.2 Code Snippet

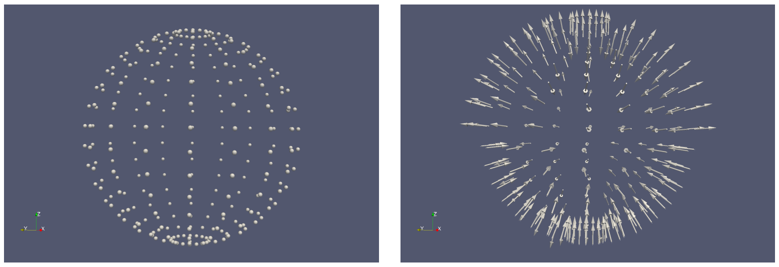
```
//obtain a polydata object containing the point set
vtkPolyData* Polydata = ....

//remove the outliers
vtkSmartPointer<vtkOutlierRemoval> OutlierRemoval =
   vtkSmartPointer<vtkOutlierRemoval>::New();
OutlierRemoval->SetInput(Polydata);
OutlierRemoval->SetPercentToRemove(.1); //specified as a value from 0 to 1
OutlierRemoval->Update();

vtkPolyData* OutputPolydata = OutlierRemoval->GetOutput();
```

3 Normal Estimation - vtkPointSetNormalEstimation

To estimate the normal at a point, we find the k nearest neighbors (specified by a parameter *kNearestNeighbors*). We find the best fit (least squares) plane through these points. The normal of this plane is taken to be the normal of the point.

3.1 Demonstration

To demonstrate the algorithm, in Figure 2 we show points on a sphere and their estimated normals.



(a) Points on a sphere (b) Estimated normals

Figure 2: Normal estimation demonstration.

Please note the inconsistency of the normal orientation. This is addressed in Section 4 by *vtkPointSetNormalOrientation*.

3.2 Code Snippet

```
//obtain a polydata object containing the point set
vtkPolyData* Polydata = ....

vtkSmartPointer<vtkPointSetNormalEstimation> NormalEstimation =
   vtkSmartPointer<vtkPointSetNormalEstimation>::New();
NormalEstimation->SetInput(Polydata);
NormalEstimation->Update();

vtkPolyData* OutputPolydata = NormalEstimation->GetOutput();
```

4 Normal Orientation - vtkPointSetNormalOrientation

There are two valid consistent orientations of surface normals - all normals facing “inside” the object, or all normals facing “outside” the object. As a good guess at “outside”, this method finds the point with the largest z value and adjusts that point's normal to point more toward the positive z direction. This initial normal direction is then propagated over the point set using the graph based technique described in the following.

We implement the technique described by Hoppe et al. in “Surface reconstruction from unorganized points”. The details are described in 4.1 and 4.2, but the overview is

- Create a Euclidean Minimum Spanning Tree (EMST) on the points.
- Add edges to the EMST to create a Riemannian graph on the points.
- Find the Minimum Spanning Tree (MST) of the Riemannian graph where the edges are weighted to promote propagation to vertices with similar normals.

- Use a heuristic to determine the "correct" orientation of a seed point and propagate this direction over the MST flipping normals that are "incorrectly" oriented.

4.1 Creating a Euclidean Minimum Spanning Tree (EMST) on a Point Set - vtkEuclideanMinimumSpanningTree

We have implemented the most naive EMST algorithm. We first create a graph with every possible edge on the points. That is, connect each point to all of the other points in the set. Set the weight of each edge equal to the distance between the two points it joins. The minimum spanning tree of this graph is called the EMST. Clearly, since n^2 edges are created, this algorithm is not scalable to large data sets. We plan to implement a more sophisticated EMST in future work.

This class is used internally by *vtkPointSetNormalOrientation*. However, it can certainly be used for other applications. The following is a simple demonstration of how to use the class.

Code Snippet

```
vtkPolyData* InputPolydata = Reader->GetOutput();

vtkSmartPointer<vtkEuclideanMinimumSpanningTree> EMSTFilter =
  vtkSmartPointer<vtkEuclideanMinimumSpanningTree>::New();
EMSTFilter->SetInput(InputPolydata);
EMSTFilter->Update();

vtkTree* EMST = EMSTFilter->GetOutput();
```

4.2 Creating a Riemannian Graph on a Point Set - vtkRiemannianGraphFilter

From the EMST, we create edges from each point to its *KNearestNeighbors* neighbors. The resulting graph is called a Riemannian graph on the points.

This class is used internally by vtkPointSetNormalOrientation. However, it can certainly be used for other applications. The following is a simple demonstration of how to use the class.

Code Snippet

```
//get the input points
vtkPolyData* input = ...

//find the Riemannian graph
vtkSmartPointer<vtkRiemannianGraphFilter> RiemannianGraphFilter =
  vtkSmartPointer<vtkRiemannianGraphFilter>::New();
RiemannianGraphFilter->SetInput(input);
RiemannianGraphFilter->SetKNearestNeighbors(5);
RiemannianGraphFilter->Update();

//get the connected graph
```

```
vtkGraph* RiemannianGraph = RiemannianGraphFilter->GetOutput();
```

4.3 Orientation/Propagation Algorithm

Once we have the Riemannian graph on the points, we set the edge weights to $w = 1 - |n_i \cdot n_j|$ where n_i and n_j are the normals of the two points joined by the edge. We then compute the MST on this graph.

Initialization

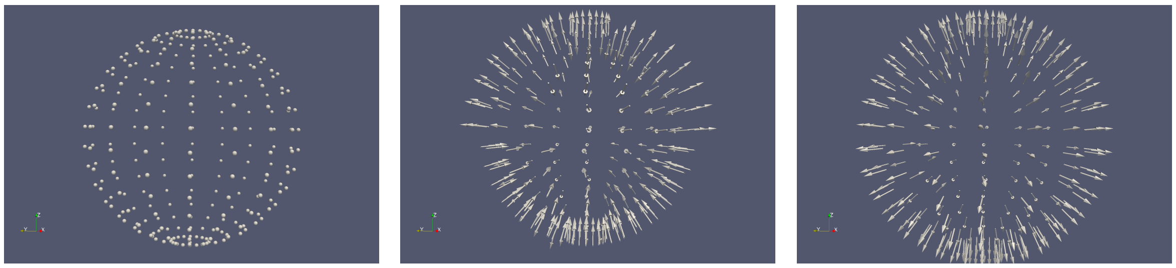
To determine the correct orientation of the normals, the point with the largest z value is found. If the dot product of the normal at this point with $(0,0,1)$ is negative, we flip the normal at this point. That is, we make the "highest" points normal point "up". This is a reasonable way to heuristically find the outside of the surface.

Propagation

We traverse the tree in a depth first fashion (when a breadth first iterator becomes available for VTK, we think this may improve the results). At each step of the traversal, consider moving from vertex i to vertex j. If $n_i \cdot n_j < 0$ (i.e. the normal of the point we are moving to is facing "away" the normal of the current point), we set $n_j = -n_j$. That is, we flip the normal to be consistently oriented.

4.4 Demonstration

To demonstrate the algorithm, Figure 3 shows an input point set, the normals produced by the normal estimation algorithm, and the correctly oriented normals.



(a) Points on a sphere (b) Estimated normals (c) Oriented normals

Figure 3: Normal orientation demonstration.

4.5 Code Snippet

```
//obtain a polydata object containing the point set
vtkPolyData* Polydata = ....

vtkSmartPointer<vtkPointSetNormalOrientation> NormalOrientation =
   vtkSmartPointer<vtkPointSetNormalOrientation>::New();
```

```
NormalOrientation->SetInput(Reader->GetOutput());
NormalOrientation->Update();

vtkPolyData* OutputPolydata = NormalOrientation->GetOutput();
```

5 Curvature Estimation - vtkPointSetCurvatureEstimation

It is often useful to know something about the “curvature” or “flatness” of a region of a point set. A reasonable indication of this idea can be produced by computing the average distance to the best fit plane of the set of points in a sphere with a chosen radius around each point. The curvature values that are computed are normalized so that the maximum value is 1. Note: This filter is not intended to approximate either Gaussian or mean curvature.

5.1 Demonstration

We show the computed curvature estimates of the points of the Stanford bunny.

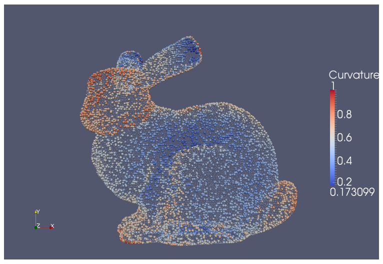


Figure 4: Curvature of the Stanford Bunny

5.2 Code Snippet

```
vtkPolyData* InputPolyData = Reader->GetOutput();

//estimate the curvature
vtkSmartPointer<vtkPointSetCurvatureEstimation> CurvatureEstimationFilter =
  vtkSmartPointer<vtkPointSetCurvatureEstimation>::New();
CurvatureEstimationFilter->SetInput(InputPolyData);
CurvatureEstimationFilter->Update();

vtkPolyData* CurvatureEstimate = CurvatureEstimationFilter->GetOutput();
```

6 Future Work

It is often necessary to compute a surface that fits the points. There are several algorithms which compute a surface given a point set. Of these, Poisson surface reconstruction is probably the most common. We intend to implement this surface reconstruction algorithm for VTK.