Douglas-Peucker Contour Simplification

Release 0.00

David Doria

July 26, 2011

Rensselaer Polytechnic Institute

Abstract

This document presents an implementation of an algorithm to find a low edge-count approximation of a complex, discrete, 2D contour. The contour can be open or closed. This implementation provides a VTK and ITK wrapper to the implementation provided by Dan Sunday at `www.softsurfer.com` (`http://www.softsurfer.com/Archive/algorithm_0205/algorithm_0205.htm`).

The code is available here: `https://github.com/daviddoria/DouglasPeuckerPolylineSimplification`

Contents

1 Introduction

This document presents an implementation of an algorithm to find a low edge-count approximation of a complex, discrete, 2D contour. Our goal is to represent the outline of an object in a simple fashion.

2 Algorithm

The algorithm is explained very well here: `http://www.softsurfer.com/Archive/algorithm_0205/algorithm_02`

We include excerpts below to make this document self contained. The algorithm consists of two steps, vertex reduction and then Douglas-Peucker simplification.

2.1 Vertex Reduction

"Vertex reduction is the brute-force algorithm for polyline simplification. For this algorithm, a polyline vertex is discarded when its distance from a prior initial vertex is less than some minimum tolerance $\varepsilon > 0$. Specifically, after fixing an initial vertex V_0, successive vertices V_i are tested and rejected if they are less than ε away from V_0. But, when a vertex is found that is further away than ε, then it is accepted as part of the new simplified polyline, and it also becomes the new initial vertex for further simplification of the polyline."

2.2 Douglas-Peucker Simplification

"Whereas vertex reduction uses closeness of vertices as a rejection criterion, the Douglas-Peucker (DP) algorithm uses the closeness of a vertex to an edge segment. This algorithm works from the top down by starting with a crude initial guess at a simplified polyline, namely the single edge joining the first and last vertices of the polyline. Then the remaining vertices are tested for closeness to that edge. If there are vertices further than a specified tolerance, $\varepsilon > 0$, away from the edge, then the vertex furthest from it is added the simplification. This creates a new guess for the simplified polyline. Using recursion, this process continues for each edge of the current guess until all vertices of the original polyline are within tolerance of the simplification."

3 Input and Output Types

The core algorithm operates on a very simple Point class (defined in point.h). A `std::vector<Point>` serves as the representation of a contour, taking the ordered points in the vector as the connectivity of the points. We provide methods to convert to and from this representation of a contour from typical VTK and ITK representations of contours.

3.1 VTK Conversions

The VTK representation of a contour is a set of ordered line segments in a vtkPolyData. To convert between the internal representation and the VTK representation, we provide the functions:

```
std::vector<Point> CreatePointVectorFromPolyData(vtkPolyData* polydata);
void WritePointsAsPolyDataLine(const std::vector<Point>& points, const std::string& fileName);
```

3.2 ITK Conversions

The ITK representation of a contour is a `itk::PolyLineParametricPath< 2 >`. To convert between the internal representation and the ITK representation, we provide the functions:

```
std::vector<Point> CreatePointVectorFromPolyLineParametricPath(itk::PolyLineParametricPath< 2
void PointsToPolyLineParametricPath(const std::vector<Point>& points, itk::PolyLineParametricP
```

3.3 Plain Text

A representation of a contour in a simple ASCII file exactly corresponds to the internal representation of a simple list of points. To read and write this type of file, we provide the functions:

```
std::vector<Point> CreatePointVectorFromPointList(const std::string& fileName);
void WritePointsAsText(const std::vector<Point>& points, const std::string& fileName);
```

4 Demonstration

A typical case of the simplification of a contour is shown in Figure 1.

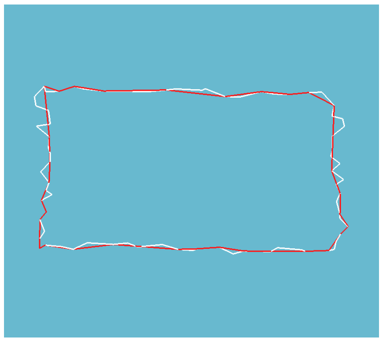

Figure 1: A demonstration of the simplification algorithm. The original contour is shown in white, where the simplified contour (tolerance = 1.0) is shown in red.

5 Code Snippet

Compilable examples are provided as DouglasPeuckerExample.cxx, DouglasPeuckerITKExample.cxx, and DouglasPeuckerVTKExample.cxx, but short excerpts are shown below to demonstrate the simple interfaces.

5.1 Plain Text Files

```
std::vector<Point> points = CreatePointVectorFromPointList(inputFileName);

// Perform the simplification
std::vector<Point> simplifiedPoints = SimplifyPolyline(approximationTolerance, points);

WritePointsAsText(simplifiedPoints, outputFileName);
```

5.2 ITK

```
itk::PolyLineParametricPath< 2 >::Pointer path = itk::PolyLineParametricPath< 2 >::New();

ReadFileIntoPolyLineParametricPath(inputFileName, path);
std::vector<Point> points = CreatePointVectorFromPolyLineParametricPath(path);

// Perform the simplification
std::vector<Point> simplifiedPoints = SimplifyPolyline(approximationTolerance, points);

WritePointsAsText(simplifiedPoints, outputFileName);
```

5.3 VTK

```
// Read the input
vtkSmartPointer<vtkXMLPolyDataReader> reader =
  vtkSmartPointer<vtkXMLPolyDataReader>::New();
reader->SetFileName(inputFileName.c_str());
reader->Update();

std::vector<Point> points = CreatePointVectorFromPolyData(reader->GetOutput());

// Perform the simplification
std::vector<Point> simplifiedPoints = SimplifyPolyline(approximationTolerance, points);

// Write the output
WritePointsAsPolyDataLine(simplifiedPoints, outputFileName);
```