A Synthetic LiDAR Scanner for VTK

Release 0.00

David Doria

September 14, 2009

Rensselaer Polytechnic Institute, Troy NY

Abstract

This document presents a set of classes (vtkLidarScanner, vtkLidarPoint, and vtkRay) to enable the production of a synthetic LiDAR scan of a 3D mesh. The scanner is implemented as a VTK filter.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3125]

Contents

In recent years, Light Detection and Ranging (LiDAR) scanners have become more prevalent in the scientific community. They capture a “2.5-D” image of a scene by sending out thousands of laser pulses and using time-of-flight calculations to determine the distance to the first reflecting surface in the scene.

Rather than setting up a collection of objects in real life and actually sending lasers into the scene, one can simply create a scene out of 3d models and ”scan” it by casting rays at the models. This is a great resource for any researchers who work with 3D model/surface/point data and LiDAR data. The synthetic scanner can be used to produce data sets for which a ground truth is known in order to ensure algorithms are behaving properly before moving to ”real” LiDAR scans. Also, noise can be added to the points to attempt to simulate a real LiDAR scan for researchers who do not have access to the very expensive equipment required to obtain real scans.

The inputs are:

- A scene to scan (triangulated 3d mesh)
- Scanner position (3D coordinate)
- Min/Max ϕ angle (how far ”up and down” the scanner should scan)
- Min/Max θ angle (how far ”left and right” the scanner should scan)
- Scanner ”forward” (the $\phi = 0$, $\theta = 0$ direction)
- Angular sample spacing or number of points to acquire in the theta and phi directions (so the ”grid” is a total of $(num_\theta \times num_\phi)$ points)

The outputs are:

- A .ptx file that maintains implicitly the structure of the scan (points are ordered as they were taken in ”strips”). This is the output given by a real Leica scanner.
- A .vtp file that is simply an unorganized point cloud of the scan returns.

1 New Classes

We introduce three new classes to implement a synthetic LiDAR scanner. The first two, vtkRay and vtkLidarPoint are supporting classes of vtkLidarScanner, which is the class that does the majority of the work.

1.1 vtkRay

This is a container class to hold a point (ray origin) and a vector (ray direction). It also contains functions to

- get a point along the ray a specified distance from the origin of the ray (double* GetPointAlong(double))

- determine if a point is in the half-space that the ray is pointing “towards“ (bool IsInfront(double*))
- transform the ray (void ApplyTransform(vtkTransform* Trans))

1.2 vtkLidarPoint

This class stores all of the information about the acquisition of a single LiDAR point. It stores

- the ray that was cast to obtain the point (vtkRay* Ray)
- the coordinate of the closest intersection of the ray with the scene (double Coordinate[3])
- the normal of the scene triangle that was intersected to produce the point (double Normal[3])
- and a boolean to determine if the ray in fact hit the scene at all (bool Hit)

1.3 vtkLidarScanner

This is the class that does all the work of acquiring the synthetic scan.

Coordinate System

The default scanner is located at (0,0,0) with orientation as follows:

- z axis = (0,0,1) = up
- y axis = (0,1,0) = forward
- x axis = (1,0,0) = right (a consequence of maintaining a right handed coordinate system)

Positioning (“aiming”) a vtkLidarScanner

A rigid transformation can be applied to the scanner via the SetTransform function which positions and orients the scanner relative to the scene.

Casting Rays

The rays cast by the scanner are relative to the frame of the scanner after the transformation is applied to the default frame.

To demonstrate the angles which must be specified, a $(\phi = 5, \theta = 4)$ scan of a flat surface was acquired, as shown in Figure 1. Throughout these examples, the red sphere indicates the scanner location, the cylinders represent the scan rays, and the blue points represent scan points (intersections of the rays with the object/scene).

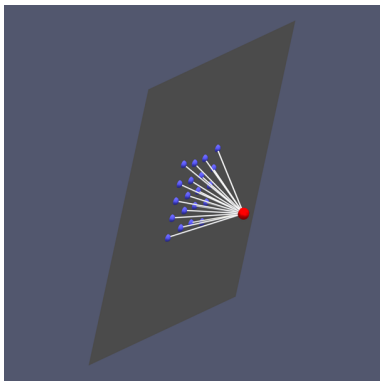


Figure 1: 3D view of the scan.

Order of Acquisition The points were acquired in the order shown in Figure 2.

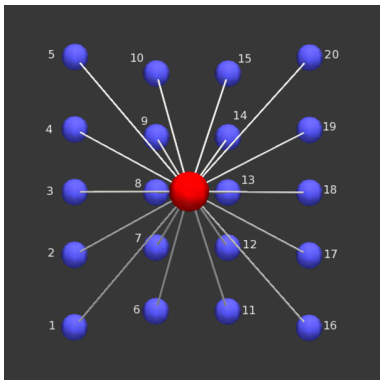
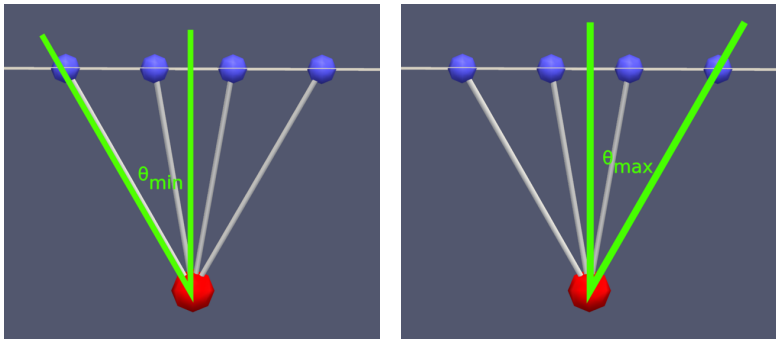


Figure 2: Scan points labeled in acquisition order.

Theta angle The angle in the Forward-Right plane (a rotation around Up), measured from Forward. It's range is $-\pi$ to π. $-\frac{\pi}{2}$ is left, $\frac{\pi}{2}$ is right.

Figure 3 shows a top view of the scan of a flat surface. The min and max Θ angles are labeled.

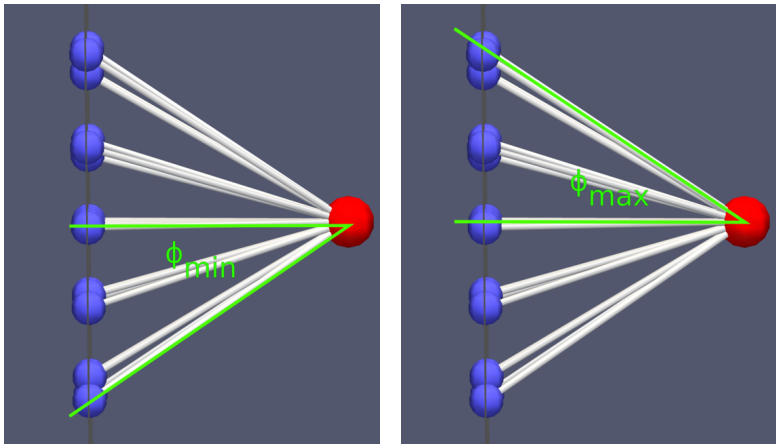


(a) Minimum Theta angle (b) Maximum Theta angle

Figure 3: Diagram of Theta angle settings.

Phi angle The elevation angle, in the Forward-Up plane (a rotation around Right), measured from Forward. It's range is $-\frac{\pi}{2}$ (down) to $\frac{\pi}{2}$ (up). This is obtained by rotating around the "right" axis (AFTER the new right axis is obtained by setting Theta).

Figure 4 shows a side (left) view of the scan of a flat surface. The min and max Φ angles are labeled.



(a) Minimum Phi angle (b) Maximum Phi angle

Figure 4: Diagram of Phi angle settings.

Normals Each LiDAR point stores the normal of the surface that it intersected. A scan of a sphere is shown in Figure 5 to demonstrate this. The normal vector coming from the scanner (red sphere) is the "up" direction of the scanner.

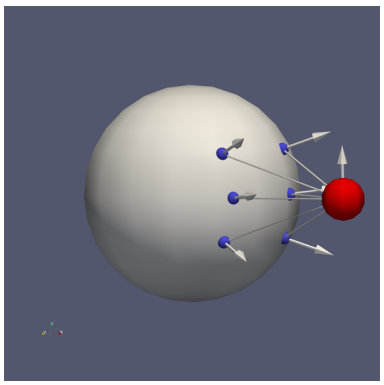


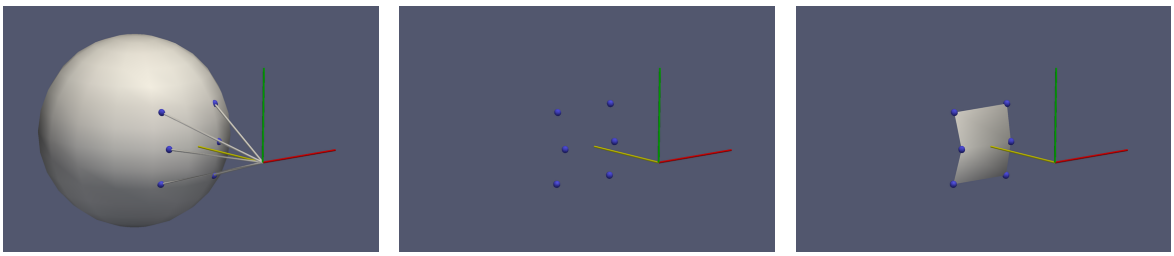
Figure 5: Scene intersections and their normals

2 Outputs

2.1 vtkPolyData

A typical output from a VTK filter is a vtkPolyData. The vtkLidarScanner filter stores the valid LiDAR returns in a vtkPolyData and return it. If the CreateMesh flag is set to true, a Delaunay triangulation is performed to create a triangle mesh from the LiDAR points. vtkDelaunay2D is used to triangulate the 3D

points utilizing the grid structure of the scan. Figure 6 shows the setup of a scan, the result with CreateMesh = false, and the result with CreateMesh = true.



(a) The scan/scene setup. (b) The resulting points. (c) The resulting points and mesh.

Figure 6: Effect of CreateMesh flag.

2.2 PTX file

All of the LiDAR ray returns, valid and invalid, are written to an ascii PTX file. The PTX format implicitly maintains the order of point acquisition by recording the points in the same order in which they were acquired. A "miss" point is recorded as a row of zeros. Upon reading the PTX file (not covered by this set of classes), the best test to see if a row of the file is valid is checking if the intensity of the return is 0. This prevents corner cases (such as a valid return from $(0,0,0)$) from creating a problem or confusion.

2.3 Scanner coordinate frame

Using `void vtkLidarScanner::WriteScanner(const std::string &Filename) const`, a .vtp file of the scanner can be written. A coordinate frame indicates the location and orientation of the scanner. The green axis is "up", the yellow axis is "forward" and the red axis is "right". Figure 7 shows the synthetic scan of a sphere along with the scanner that was used to perform the scan.

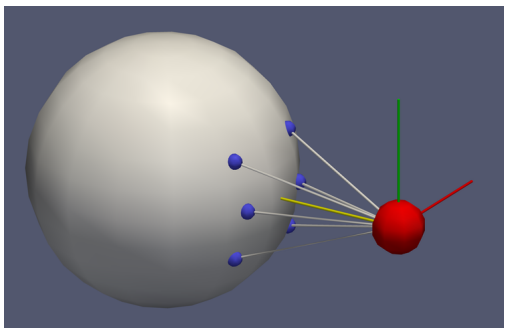


Figure 7: Synthetic scan of a sphere with the scanner displayed

3 Data structure speed up

Instead of intersecting each ray with every triangle in the scene, this problem immediately lends itself to using a spatial data structure to achieve an enormous speedup. We originally tried an octree (vtkOBBTree),

but I found that a modified BSP tree (vtkModifiedBSPTree) gives a 45x speedup even over the octree! The current implementation includes this speed up and is therefore very fast.

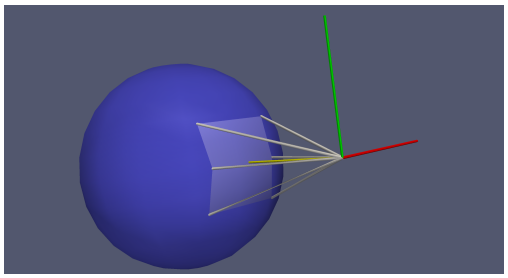


Figure 8: A noiseless synthetic scan

4 Noise Model

By default, a synthetic scan is "perfect" in the sense that the scan points actually lie on a surface of the 3D model as in Figure 8. In a real world scan, however, this is clearly not the case. To make the synthetic scans more realistic, we have modeled the noise in a LiDAR scan using two independent noise sources: line-of-sight and orthogonal.

4.1 Line-of-Sight (LOS) Noise

This type of noise is error in the distance measurement performed by the scanner. It is a vector parallel to the scanner ray whose length is chosen randomly from a Gaussian distribution. This distribution is zero mean and has a user specified variance (double LOSVariance). An example of a synthetic scan with LOS noise added is shown in 9. The important note is that the orange (noisy) rays are exactly aligned with the gray (noiseless) rays.

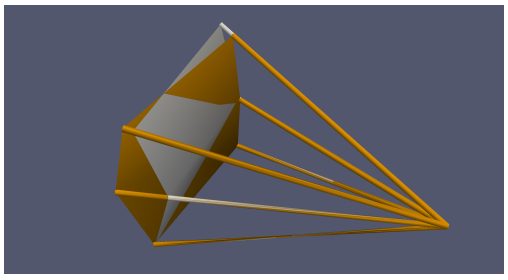


Figure 9: A synthetic scan with line-of-sight noise added

4.2 Orthogonal Noise

This type of noise models the angular error of the scanner. It is implemented by generating a vector orthogonal to the scanner ray whose length is chosen from a Gaussian distribution. This distribution is also zero mean and has a user specified variance (double OrthogonalVariance). An example of a synthetic scan with orthogonal noise added is shown in 10. Note that the green (noisy) rays are not aligned with the grey (noiseless) rays, but they are the same length.

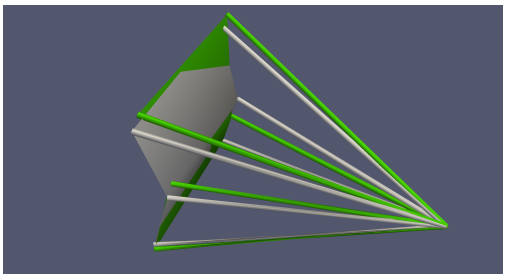


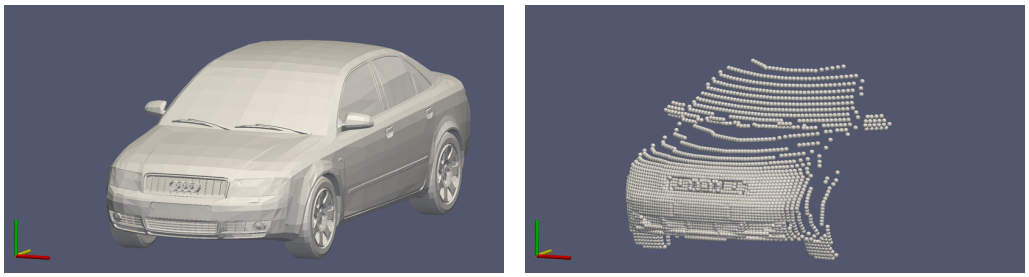
Figure 10: A synthetic scan with orthogonal noise added

4.3 Combined Noise

A simple vector sum is used to combine the orthogonal noise vector with the LOS noise vector.

5 Example Scene

As an example, a car model with 20k triangles was scanned with a 100x100 grid. On a P4 3GHz machine with 2gb of ram, the scan took 0.6 seconds. Figure 11 shows the model and the resulting synthetic scan.



(a) A car model (b) A synthetic scan of the car model

Figure 11: A car model and the resulting synthetic scan.

6 Example Code

An example (TestScanner.cpp) is provided with the code zip file, but the basics are demonstrated here below with hard coded values:

```
//read a scene
vtkXMLPolyDataReader* reader = vtkXMLPolyDataReader::New();
reader->SetFileName(InputFilename.c_str());
reader->Update();

//construct a vtkLidarScanner
vtkLidarScanner* Scanner = vtkLidarScanner::New();
```

```
  //Set all of the scanner parameters

Scanner->SetPhiSpan(vtkMath::Pi()/4.0);
Scanner->SetThetaSpan(vtkMath::Pi()/4.0);

Scanner->SetNumberOfThetaPoints(5);
Scanner->SetNumberOfPhiPoints(6);

Scanner->SetStoreRays(true);

//"aim" the scanner.  This is a very simple translation, but any transformation will work
vtkTransform* transform = vtkTransform::New();
transform->PostMultiply(); //this is so we can specify the operations in the order they should

//(any of these 4 lines can be ommited if they are not required)
transform->RotateX(0.0);
transform->RotateY(0.0);
transform->RotateZ(1.0);
transform->Translate(Tx, Ty, Tz);

Scanner->SetTransform(transform);

//indicate to use uniform spherical spacing
Scanner->MakeSphericalGrid();

Scanner->SetCreateMesh(true);

Scanner->SetInput(reader->GetOutput());
Scanner->Update();

//create a writer and write the output vtp file
vtkSmartPointer<vtkXMLPolyDataWriter> writer = vtkSmartPointer<vtkXMLPolyDataWriter>::New();
writer->SetFileName("test_scan.vtp");
writer->SetInput(Scanner->GetOutput());
writer->Write();
```