Normal estimations in Inviwo

Ingemar Markström

February 5, 2019

Why this thesis?

3D Graphics crash course

Vectors

- A vector $\vec{v} = (x, y, z)$ is:
  - ... a coordinate somewhere in space.
  - ... the direction towards a point in space.
- Traveling from $\vec{v_1}$ to another $\vec{v_2}$ creates a line.

Planes and normals

- Three non-equal points $[\vec{v_1}, \vec{v_2}, \vec{v_3}]$ make two lines:
  1. $\vec{t_1} = \vec{v_2} - \vec{v_1}$
  2. $\vec{t_2} = \vec{v_3} - \vec{v_1}$
- All points $\vec{p} = a * \vec{t_1} + b * \vec{t_2}$ outline a plane.
- The normal of the plane is the cross product $\vec{n} = \vec{t_1} \times \vec{t_2}$.

Small demo

Unorganized point cloud?

- A collection of vectors $P = [\vec{v_1}, \vec{v_2}, ..., \vec{v_n}]$.
- No structure.
- No connectivity.

Point clouds in this thesis

Point clouds originating from:

- Visionair Aim@Shape Digital Shape Workbench.
- Stanford Computer Graphics Laboratory.
- My own creations.

What is Inviwo?

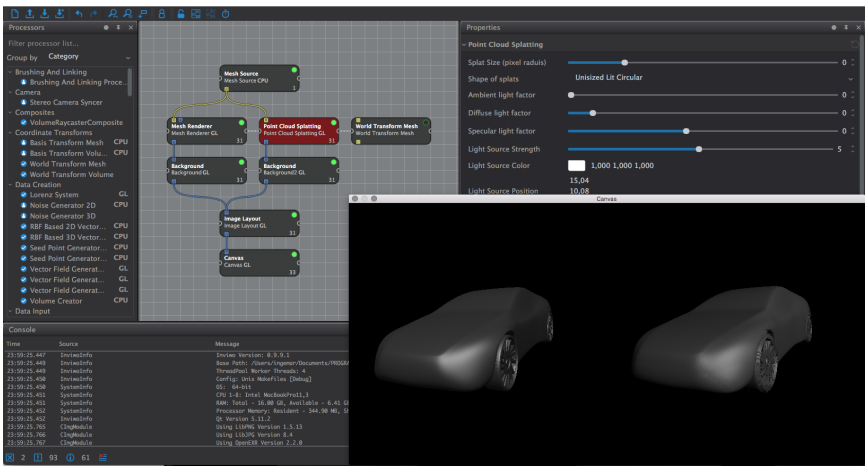
In short

- Open source scientific visualization framework.
- Processors and modules written in C++, using OpenGL/Vulcan, OpenCL and OpenMP.

Can visualize

- Geometry (Meshes, lines etc)
- Scalar fields (images, volumes)
- Vector field (streams, paths etc)

Example of usage



Objective of this thesis

Inviwo lacked:

- A useful point splatting module.
- A normal estimation processor for unorganized point clouds.

Research question

How do different algorithms and datastructures for estimating normals in unorganized point clouds compare in terms of output quality and calculation time?

Goal

- Implement a usable normal estimation and rendering solution for unorganized point clouds in Inviwo.

Why a new point splatting processor?

I wanted:

- Color-coded debug output of estimated normals.
- Possible use of at least one light source.

Normal estimation evaluation

- Output quality:
  - Visual image inspection compared to references.
  - Numerical error distribution analysis.
- Estimation running-time and resources
  - Timing of each step in the estimation process.
  - Static analysis of memory usage.

Point splatting

TODO Video

Reasons

- To few points to cover enough screen surface.
- Down-sampling for increased calculation performance.

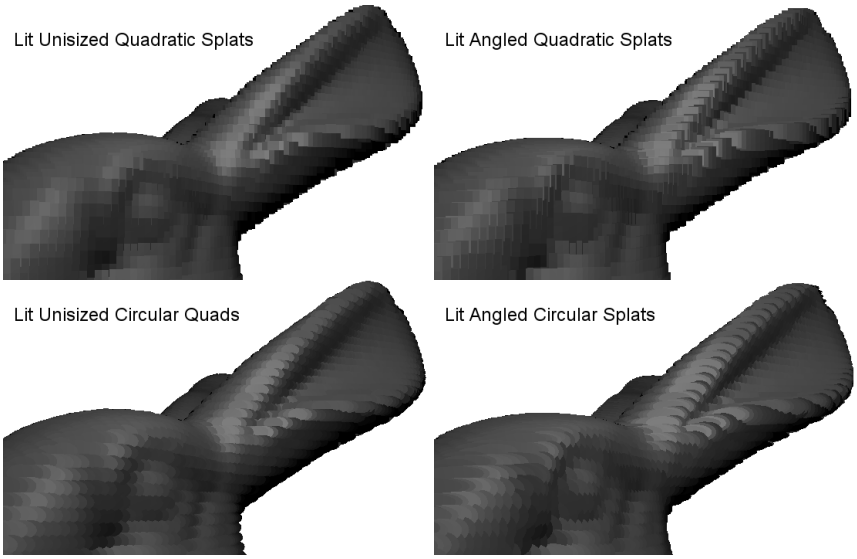
TODO How

Two images:

1. Only point
2. Point, plus four other points forming a splat.

Variations

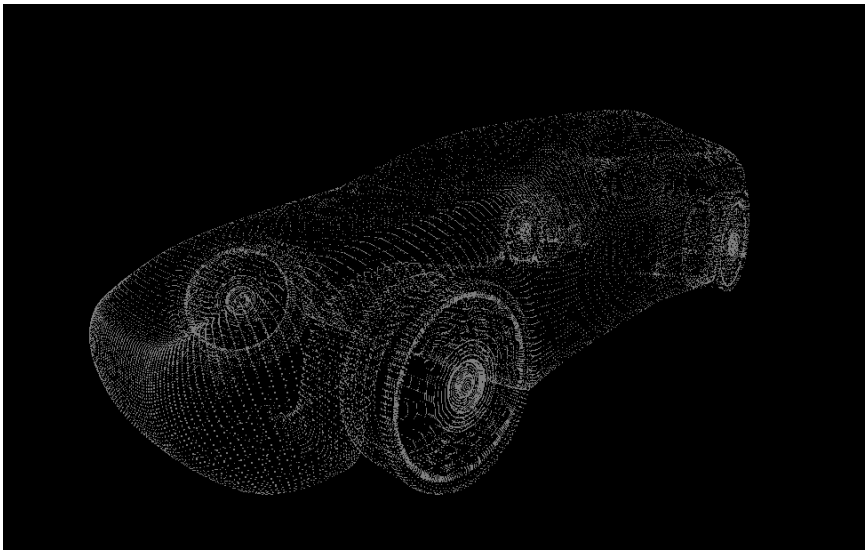
- Different shapes.
- Placements offsets and angle.
- Adaptive size to neighboring points.



Normal estimation

What we see on screen

A car:



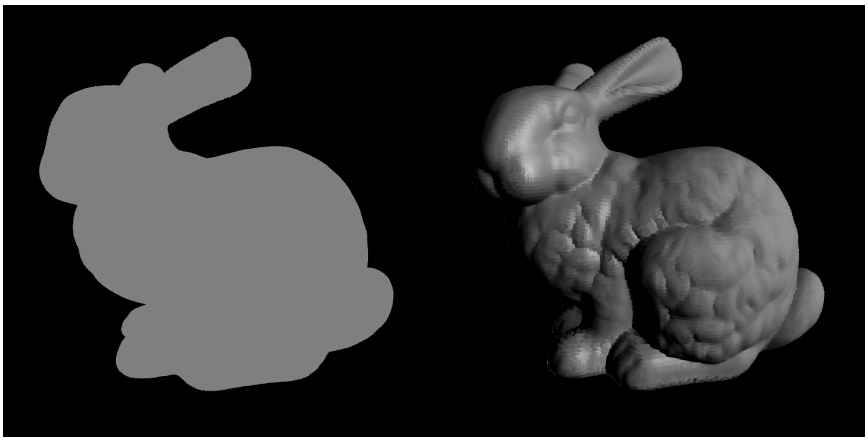
What the computer see

The same car:

```
.
.
..
...
0.512561 -0.076571 2.188105
0.514271 -0.078355 2.189595
0.516147 -0.080143 2.190959
0.518189 -0.081957 2.192225
0.520399 -0.083819 2.193421
0.522778 -0.085751 2.194574
0.525326 -0.087774 2.195714
0.504853 -0.083938 2.202324
0.506327 -0.085852 2.203986
0.507973 -0.087758 2.205488
0.509786 -0.089681 2.206867
0.511765 -0.091646 2.208155
0.513907 -0.093677 2.209388
0.516210 -0.095799 2.210600
0.497435 -0.090691 2.215300
0.498684 -0.092730 2.217127
0.500109 -0.094749 2.218763
0.501706 -0.096775 2.220250
0.503466 -0.098836 2.221627
0.505385 -0.100960 2.222934
0.507456 -0.103175 2.224214
0.197373 -0.124149 2.321243
0.219923 -0.124149 2.319959
0.241723 -0.124149 2.318556
```

Why estimate normals?

- Needed for proper light calculations.
- Useful in computer vision and object recognition.



Different approaches to normal estimation from neighborhoods

The two categories are:

1. Averaging methods.
2. Optimization methods.

And the process can be described summarized as:

1. For each point $\vec{p_i}$, find other nearby points $E = [e_{i,1}, e_{i,2}, ..., e_{i,k}]$.
2. Choose and use one of the methods above.

Finding K point neighbors in an unorganized point cloud

- Both estimation categories utilize local neighborhoods.
- Linear search is painfully slow ($O(n^2 * k)$)
- Restructure the points into a tree structure. Which one?
  - Oct-tree:
    - Often used in game engines.
    - Fast creation.

        * Fast collision lookup, if the sub-tree searched from the root.
    - Equally spaced voxel grid:
        * Intuitive neighboring voxel traversal.
        * However, possibly many empty voxels.
    - KD-tree:
        * Balanced tree.
        * Slower initialization than the oct-tree.
        * Intuitive nearest neighbor querying.

KD-Tree

Worth remembering:

- The unorganized point cloud lacks any specific ordering.
- Reordering points does not alter the visual output from the point splatting processor.

Building a 3D KD-tree from an unorganized point cloud $P = [p_{\vec{start}}, p_{\vec{start+1}}, ..., p_{\vec{end}}]$:

1. Find the median point p_m in a dimension $d \in \{x, y, z\}$ among all points in P.
2. Put all points with $x_i < p_m$ before, and the rest after the median
3. Create two sub-trees (if there are any points left):
    - $t_{left} \rightarrow$ points $[p_{\vec{start}}, ..., p_{\vec{m-1}}]$.
    - $t_{right} \rightarrow$ points $[p_{\vec{m+1}}, ..., p_{\vec{end}}]$.
4. Start over from 1) for each sub-tree in another dimension.

Nearest neighbor search

- Two main types of neighborhoods:
    - Fixed size.
    - All neighbors in a fixed radius.
- The neighborhood search differs only in when to add-conditional.

Averaging methods

Variations

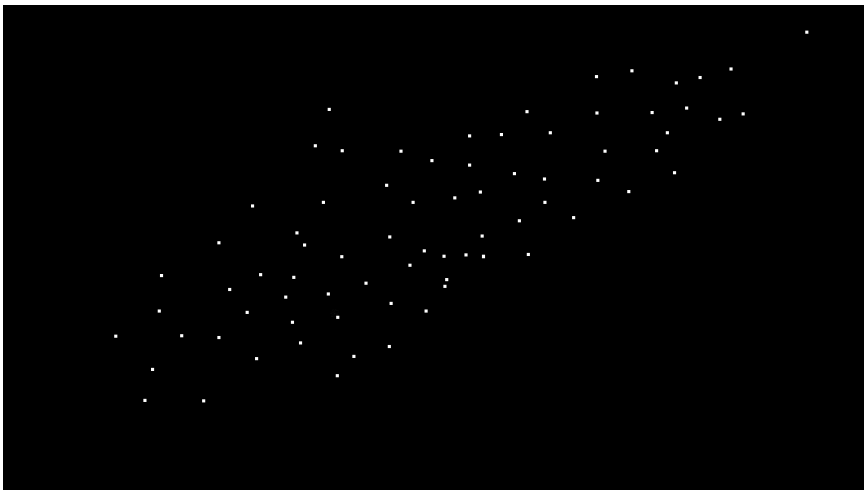
- The number of triangles formed
  - Fandisk (as in the video).
  - Complete triangulation of all neighbors.
- Weighting the different triangles formed.
  - Triangle edge length.
  - Triangle area.

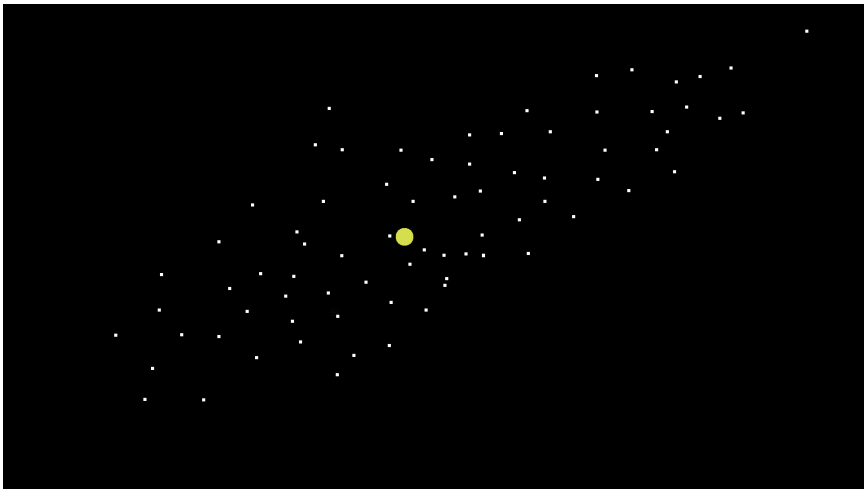
Numerical methods (PCA)

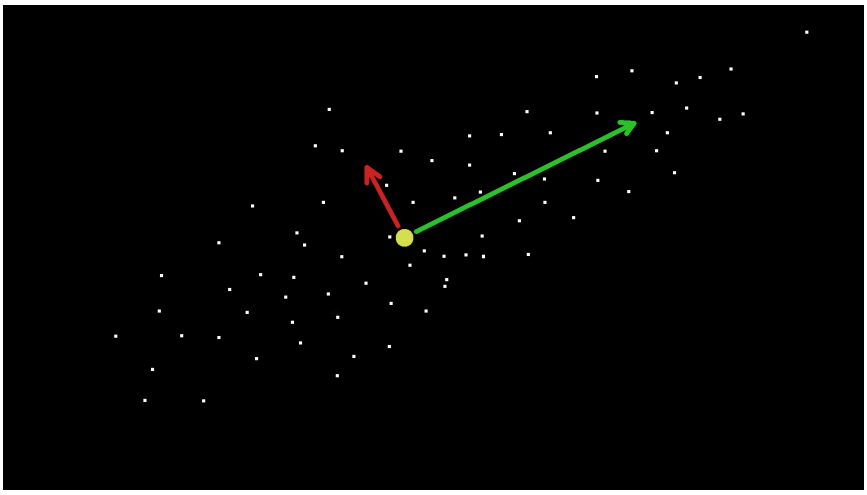
Principal Component Analysis:

1. Find the nearest neighbours.
2. Create a covariance matrix M of the neighborhood.
3. Find the eigenvectors and eigenvalues of M.
4. The smallest eigenvalue correspond to the neighborhood normal.

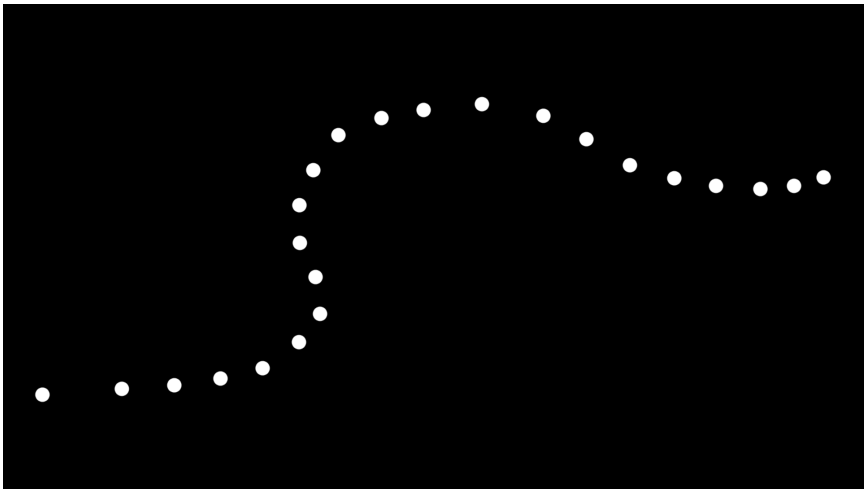
Short standard image example in 2D

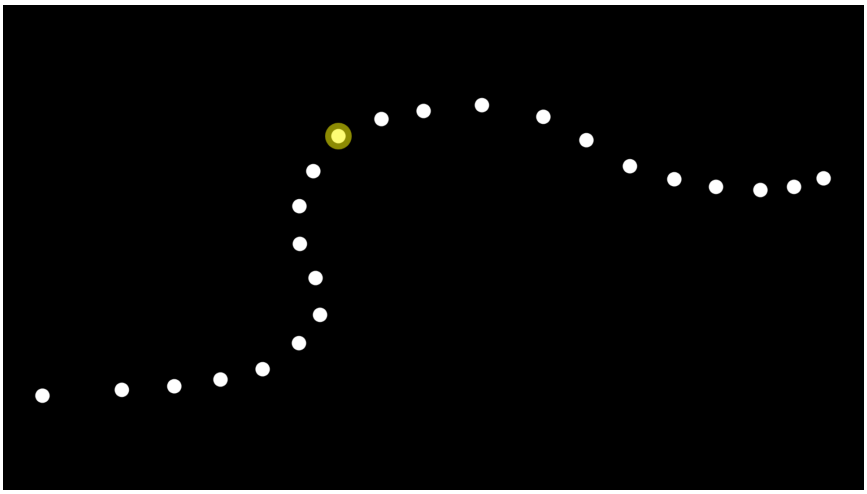


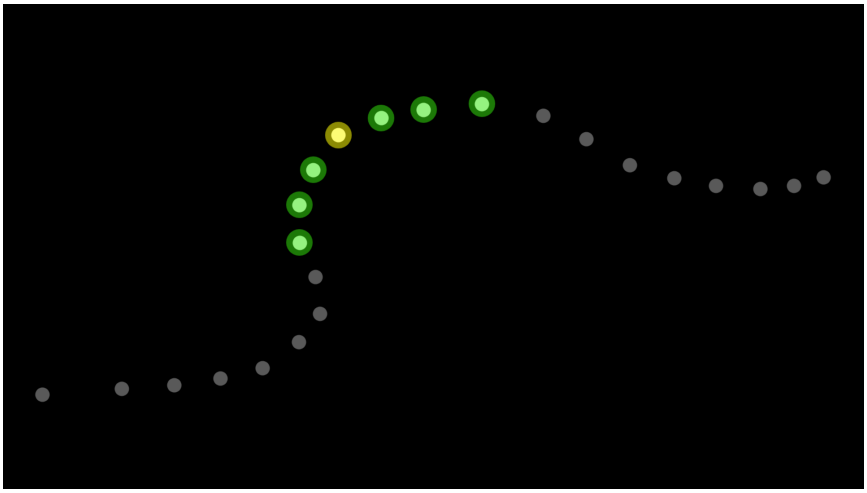


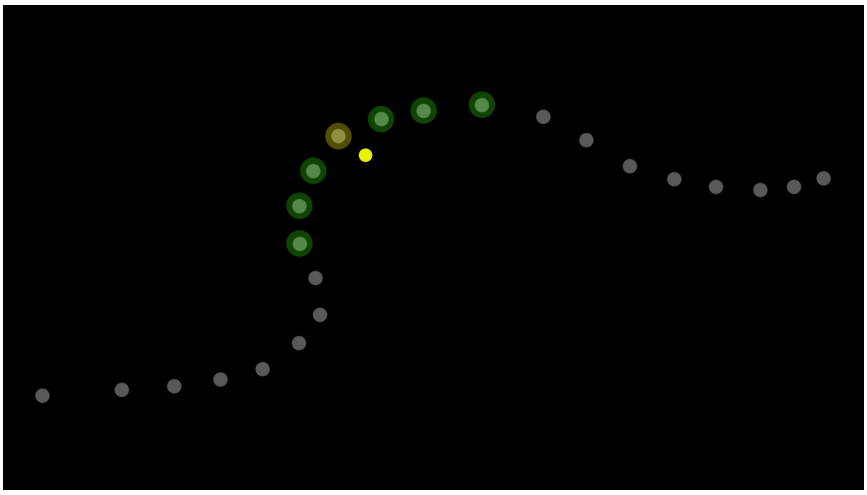


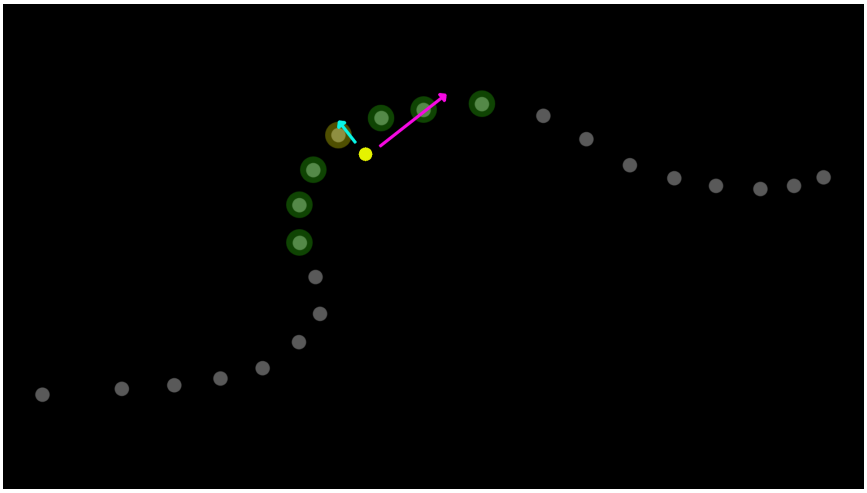
When done on a 2D curve











For a more explicit outline of the maths, see the report, section 2.3.2.