Data Visualization (CIS/DSC 468)

Data

Dr. David Koop
Overview

Reducing Items and Attributes

- **Filter**
  - Items
  - Attributes

- **Aggregate**
  - Items
  - Attributes

**Reduce**

- Filter
  - Items
  - Attributes

- Aggregate
  - Items
  - Attributes

- Embed

[Munzner (ill. Maguire), 2014]
K-Means

Run

[C. Polis, 2014]
Tasks in Understanding High-Dim. Data

**Task 1**
- **In**: HD data
- **Out**: 2D data
- **What?**
  - In High-dimensional data
  - Out 2D data
- **Why?**
  - Produce
  - Derive

**Task 2**
- **In**: 2D data
- **Out**: Scatterplot Clusters & points
- **What?**
  - In 2D data
  - Out Scatterplot
- **Why?**
  - Discover
  - Explore
- **How?**
  - Identify
  - Navigate
  - Select

**Task 3**
- **In**: Scatterplot Clusters & points
- **Out**: Labels for clusters
- **What?**
  - In Scatterplot
- **Why?**
  - Produce
- **Annotate**

[Munzner (ill. Maguire), 2014]
Principle Component Analysis (PCA)

Original data space vs. component space after PCA.

[Gene 1, Gene 2, Gene 3]

PC 1 vs. PC 2

[M. Scholz, CC-BY-SA 2.0]
Focus+Content Overview

Embed

Elide Data

Superimpose Layer

Distort Geometry

Reduce

Filter

Aggregate

Embed

[Munzner (ill. Maguire), 2014]
Elision: DOI Trees

- Example: 600,000 node tree
  - Multiple foci (from search results or via user selection)
  - Distance computed topologically (levels, not geometric)

[Heer and Card, 2004]
Degree of Interest Function

- DOI = I(x) - D(x,y)
  - I: interest function
  - D: distance (semantic or spatial)
  - x: location of item
  - y: current focus point (could be more than one)
- Interactive: y changes
Superimposition with Interactive Lenses

(a) Alteration

(b) Suppression

[ChronoLenses and Sampling Lens in Tominski et al., 2014]
It can be difficult to observe micro and macro features simultaneously with complex graphs. If you zoom in for detail, the graph is too big to view in its entirety. If you zoom out to see the overall structure, small details are lost.

Focus + context techniques allow interactive exploration of an area.

Distortion

[M. Bostock, http://bost.ocks.org/mike/fisheye/]
Distortion Concerns

• Distance and length judgments are **harder**
  - Example: Mac OS X Dock with Magnification
  - Spatial position of items changes as the focus changes
• Node-link diagrams not an issue… why?
• Users have to be made aware of distortion
  - Back to scatterplot with distortion example
  - Lenses or shading give clues to users

• **Object constancy**: understanding when two views/frames show the same object
  - What happens under distortion?
  - 3D Perspective is distortion… but we are well-trained for that
• Think about **what** is being shown (filtering) and method (fisheye)
Focus+Context in Graph Exploration

(a) Moderately large graph drawn with straight line edges. The graph nodes correspond to the USA major cities; edges show migration flows. The graph contains 1715 nodes and 9778 edges. Nodes are laid out according to geographical positions of cities, producing a drawing with poor readability, where edges mix in a totally unordered way and where some nodes are close to unnoticeable.

(b) The same graph as in Fig. 1(a) now drawn using edge bundling with edges rendered as B´ezier curves.

Figure 1: Illustration of edge bundling.

(a) The fish-eye distorts a small region of the graph for local inspection.
(b) The magnifying lens shows a zoom on a local region.

Figure 2: Fisheye and magnifying lens

Magnifying Lens and Fish-eye – The magnifying lens [3] and geometrical fish-eye [7] were also added to the system as basic interactors. They allow to get local details on an area of the graph without having to zoom in (see Fig. 2(a) and Fig. 2(b)). These techniques allow to get an overall estimate on the degree of nodes or number of edges that have been bundled together, and an idea on the spatial organization of neighborhoods.

Neighborhood highlighting – After edges have been bundled, the graph gains in overall readability at the loss of more local information. For instance, connections between any two particular nodes cannot be easily recovered and isolated out of a bundle. When designing the system and deciding on the interactions to implement and combine, we focused on the recovery of these local information. By hovering the mouse over any node in the graph drawing, the user can highlight its neighborhood. This is accomplished by showing a translucent circle over the immediate area where a node sits while clearly displaying the neighborhood of the node (top of Fig. 3(a)). The circle fades off nodes not belonging to the selected neighborhood, temporarily providing a clear view of it. The size of the translucent circle is fitted as to enclose all immediate neighbors of the node in the graph. Using the mouse wheel, the user can select neighbors sitting at a bounded distance from the node. The size of the translucent circle adjusts accordingly (bottom of Fig. 3(b)).

Bring & Go – Now, neighborhood nodes in the graph don’t always sit close. As a consequence, the translucent circle highlighting neighbors of a node can potentially be quite large. That is, the distance between nodes in the graph does not always match their Euclidean distance in the drawing.

Focus+Context in Graph Exploration

[Source: Lambert et al., 2010]
(a) Neighborhood highlighting – selecting a node brings up its neighbors, fading away all other graph elements.

(b) Using the mouse wheel, the neighborhood is extended to nodes sitting further away.

Figure 3: Illustration of the Neighborhood highlighting in interaction this indeed is the challenge posed to all layout algorithms.

The Bring & Go technique introduced by Tominski et al. [18] solves this paradox. The Bring operation pulls neighbors of a node to near proximity, temporarily resolving a situation where the layout algorithm had failed. Fig. 4(a) and Fig. 4(b) illustrates this situation – the passage from step 1 to step 2 being smoothly animated. Once the neighbors have been repositioned close to the node, the Go operation lets the user decide of a new direction to move to by selecting a neighbor. After clicking a neighbor node, the visualization is panned until re-centered around the target neighbor. The transition is performed by smoothly animating the pan (see Fig. 3). A recent user-study of this interaction technique has been made by Moscovich et al. [15]. When bringing neighbors close to the selected node, the edges abandon their curve shapes and are morphed to straight lines. This is done by modifying the control points coordinates of each curve so that they are all aligned.

Our system thus comprises a comprehensive palette of interactions focusing on adjacency or accessibility tasks (we borrow this terminology from Lee et al.’s [14] task taxonomy, itself referring to the work of Amar et al. [1]). That is, tasks such as exploring neighbor nodes, or counting them, finding how many nodes can be accessed from any given one, etc., can be easily done through direct manipulation of the graph using zoom, pan, neighborhood highlight or Bring & Go, for instance. All these interaction techniques have been implemented as interactor plugins for the Tulip graph visualization software [2] and are available through its plugin server.

4 Maintaining fluid interaction

The challenge we were faced with is that curves generation have a relatively high computational cost when it comes to interacting with bundles. Indeed, although the curves can be drawn in reasonable time for static drawings using standard rendering techniques, the problem becomes tedious when one wants to interact on bundles using any of the techniques described in the previous section. The curves’ shapes must be continually transformed as the user moves the mouse and pilots interaction (geometrical fisheye or Bring & Go for instance).

Moreover, we did not want fluidity to impact on the quality of the curves and impose an upper bound on the number of control points used to compute the edge routes. Instead, we aimed at producing a system capable of dealing with an arbitrary number of control points. As a consequence, the computation of the points interpolating the curve itself puts a real burden on the system and calls for an extremely efficient approach. The solution we designed avoids performing computations on the CPU as far as possible, relying on the GPU for almost all curve related computations. The only computations that are potentially performed on the CPU are the original graph layout and the bundling part.

4.1 Introduction to spline rendering

Now, there are two major issues when rendering a parametric spline. Control points define the curve analytically described as a polynomial (see Eq. (1 for Bézier curves). Second, once the polynomial has been determined, it must be evaluated as many times as required in order to interpolate the curve itself. As a consequence, when interacting with the graph asking for local deformation of edges, bringing neighbors closer or following an edge, the curves must be re-computed on the fly.

A classical approach when rendering a curve is to compute the interpolation points on the CPU, then call appropriate graphics primitives and let the GPU render the curve.
Focus+Context in Graph Exploration

(a) Bring (step 1) – Selecting a node fades out all graph elements but the node neighborhood.
(b) Bring (step 2) – Neighbor nodes are pulled close to the selected node.
(c) Go – After selecting a neighbor (the green node in Fig. 4(b)), a short animation brings the focus towards a new neighborhood.

Figure 4: Illustration of the Bring & Go interaction.

In order to be able to easily interact with the edge-bundled graphs, even for basic interactions like panning and zooming, we have to optimize the curves rendering by reducing the computational load on the CPU as much as possible. One solution could be to pre-compute all curve points and store them in memory; this obviously is not efficient in terms of memory usage, considering that we want to draw a large amount of fine-grained rendered curves. For example, drawing $10^5$ curves (edges) with 100 points per curves – one point being stored as 3 floats (4 bytes each), the total amount of memory use would be $\sim 10^8$ bytes (more than 110 Mbytes).

Another solution will be to use the built-in components of high level graphics API for rendering curves. For instance, in OpenGL, that task can be achieved by using a standard feature called evaluators. Evaluators can be used to construct curves and surfaces based on the Bernstein basis polynomials. This includes B´ezier curves and patches, and B-splines. An evaluator is set up from an array of control points and allows to compute curve points on the GPU by sending the parameter $t$ to the rendering pipeline. However, most of the OpenGL implementations have restrained the maximum authorized number of control points to eight. So to draw a B´ezier curve or a cubic B-spline with more than eight control points using evaluators, it has to be done piecewise by subdividing the curve to render into curves with fewer control points. Consequently, the performance to draw high order curves with this technique decreases as the number of control points grows. So even if evaluators work well to render curves with a small number of control points, they are not suitable to resolve our issue of drawing curves with several dozens of control points efficiently.

4.2 GPU-intensive spline rendering

Our solution delegates the computation of curve points to the GPU which is perfectly well designed to perform vectorial computation and floating points operations. By using the OpenGL Graphics API, we can encapsulate those tasks in a shader program. This type of program, written in a C-like language called GLSL (OpenGL Shading Language), allows to modify the default behavior of some processing units in the rendering pipeline – the vertex processing unit can be customized this way. The purpose of vertex processing stage is to transform each vertex's 3D position in virtual space to the 2D coordinates at which it appears on the screen. By designing a vertex shader we can manipulate properties such as node position or color, with all computations executed on the GPU. Shaders offer tangible benefits since they are well suited for parallel processing as most modern GPUs have multiple shader pipelines. The vertex shader we designed is activated each time we render a curve on screen. Before sending vertex coordinates to the GPU, the curve's control points are transferred to the shader and stored in an array. The maximum size of that array is hardware dependent and determined at runtime. On recent GPU, more than one thousand control points are passed to the shader program.

[Lambert et al., 2010]
Assignment 4

• Interaction!
Data Wrangling

• Problem 1: Visualizations need data
  • Solution: The Web!

• Problem 2: Data has extra information I don't need
  • Solution: Filter it

• Problem 3: Data is dirty
  • Solution: Clean it up

• Problem 4: Data isn't in the same place
  • Solution: Combine data from different sources

• Problem 5: Data isn't structured correctly
  • Solution: Reorder, map, and nest it
Hosting data

- github.com
- gist.github.com
- myjson.com
- Google Drive
- Other services
Why JavaScript?

- Python and R have great support for this sort of processing
- Data comes from the Web, want to put visualizations on the Web
- Sometimes unnecessary to download, process, and upload!
- More tools are helping JavaScript become a better language
Online JavaScript Resources

- [http://learnjsdata.com/](http://learnjsdata.com/)
- Good coverage of data wrangling using JavaScript
Data Wrangling Example: 
Census Data by State