Applied Data Visualization in Virtual Reality

Jed Tan, Steve Kim

June 6, 2016

1 Introduction

With Virtual Reality (VR) hardware becoming increasingly used in business and industry, processing and exploring Big Data in VR is a natural following step. Integrating data visualization techniques with VR technology poses numerous challenges, but one we hope to solve is allowing real-time exploration and processing of large datasets in virtual reality. This can be broken down into 3 smaller questions. Which hardware is best for interacting with visualizations in VR? Secondly, how should we design user interface in conjunction with our choice of hardware? Lastly, how should we design specific instances of visualizations for use in VR? To attempt to solve this problem, we attempt to create a system that produces automatically generated interactive visualizations for use with VR hardware. To narrow the scope of the project, we specifically focused on implementations of network graph visualizations, for which there exists established algorithms for construction. The end result is a functioning prototype of a data visualization system able to take raw network data and convert it automatically to a VR-optimized interactive visualization.

2 Previous Work

2.1 3-D Data Visualization Construction

Previous work for creating 3-D Data Visualization yields interesting ideas for automated generating of visualizations. Takatsuka and Wu's paper "Visualizing multivariate network on the surface of a sphere"[4] gave us the base algorithm with which we generated mappings of network nodes and edges on to spherical structures. Their previous work to create 3-D visualizations while encoding information associated with multivariate data resonated strongly with our goals. Kwon et al.'s work with B-Splines [3] in similar three dimensional spaces gave us suggestions as to potential future work for visualizations in a 3-D space.

2.2 Data Visualization in VR

We were surprised to see the lack of academic literature on data visualization in VR, though many companies are known to be competing in the space. Most current implementations require rooms with motion sensors where the entire room has motion detectors, such as the CAVE system [2]. However, we noticed that most existing implementations we see do not use 3D objects and structures to represent data. Much like the figure below, many implementations of data visualization in VR simply display 2D screens of data visualizations within a 3D space. While this approach may provide some advantages, the purpose of this research is to create visualizations that could take advantage of the extra spatial dimension.

3 Methods

3.1 Preprocessing

Several data sets were downloaded and prepared for use in the space filling algorithm. As the python script for conversion took as input a list of nodes and a dictionary representing an adjacency list, we used a mixed process of hand-converting data sets and writing simple parsers to turn data set .txt files to adjacency lists readable by the algorithm-executing script.

3.2 Space Filling Algorithm

The algorithm for converting lists of edges and nodes to a 3-D mapping of edges and nodes in real space was done using a modified version of the Yingxin and Takatsuka algorithm for mapping nodes and edges to a sphere. The first step of the algorithm is to randomly select points on the surface of a sphere using the Marsaglia algorithm. First, two floating point numbers are sampled from a uniform distribution over the values (from herein referred to as x_1 and x_2). Values of x_1 and x_2 such that $x_1^2 + x_2^2 \geq 1$ are rejected. Afterwards, x, y and z coordinates are generated via the following formulas:

$$x = 2x_1\sqrt{1 - x_1^2 - x_2^2}$$

$$y = 2x_2\sqrt{1 - x_1^2 - x_2^2}$$

$$z = 1 - 2(x_1^2 + x_2^2)$$

Next, to optimize the physical location of the nodes, a perturbation and scoring algorithm is applied to each of the nodes for n iterations (n being a tunable parameter based on time available and experimentation). The perturbation proceeds as follows:

1. For each location a node is placed, create a list of eight perturbation points located around the node.

2. For each point around the node, calculate both an edge-edge crossing score EC and a edge-node proximity score EP. These two sum to a full score $S = EC + EP$. The formula for calculating these scores follows.

3. Assign the new position of the node as the perturbation point around the current node that had the lowest full score S.

3.2.1 Edge-Edge Crossing Score

This score was designed to penalize having edges crossing over each other - crossing edges can be distracting to a user and reduces a user's ability to read a graph. Thus, a node was assigned a flat penalty for each collision between one of it's outgoing edges and another edge between other nodes. Calculating if two edges collide is done with the following algorithm (adapted from Yingxin and Takatsuka):

Given edges from node A to B and node C to D, and the origin as O, we first calculate normal vectors:

$$\vec{N_1} = \vec{OA} \times \vec{OB}$$

$$\vec{N_2} = \vec{OC} \times \vec{OD}$$

These two vectors thus define a plane that is perpendicular to the vector where edges AB and CD would overlap. Thus, we calculate the vector from the origin to the two possible intersection points E and F as:

$$\vec{OE} = \frac{\vec{N_1} \times \vec{N_2}}{\|\vec{N_1} \times \vec{N_2}\|}$$

$$\vec{OF} = -\vec{OE}$$

If the great-circle length $\|MF\|$ or $\|ME\|$ is found to be less than half the length of AB and CD, the two edges must intersect. These edge lengths can be calculated via the formula $L = arccos(A\dot{B})$ Reference Fig 1 for a visual explanation of this phenomenon.

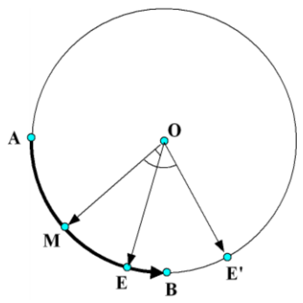

Figure 1: Visual Explanation of Edge-Crossing: Note that if length EM < BM, point E is on the edge AB

3.2.2 Edge-Node Crossing Score

This score was designed to penalize having edges too close to a node. A user confusing which node a particular edge originates from can be disorienting as well. Because this problem was perceived as being much more detrimental (and theoretically rare) to the visibility of a graph, the score penalty for having an edge too close to a node was much higher than that of having edges cross. Calculation of edge-node distance was done with the following algorithm:

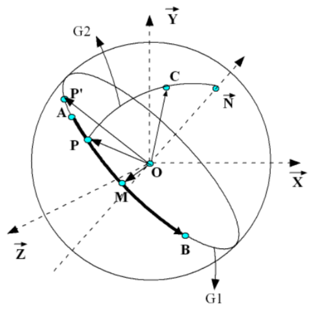

Figure 2: Visual Explanation of Edge-Node Distance

Given edges from node A to B, origin at O and a node at position C, firstly find the normal vector to the plane AOB via the formula $\vec{N} = \vec{OA} \times \vec{OB}$. The great circle containing both N and point C must contain the closest point between edge AB and C. After normalizing N, we can then find the closest points on the edge similarly to the previous algorithm for the calculation of points E and F (represented by point P in figure 2), instead replacing $\vec{N_1}$ and $\vec{N_2}$ with $\vec{NC}$ and $\vec{AB}$. The arc length of CE and CF can then be found using the arccos formula mentioned in section 3.2.2. If one of these distances is below a certain threshold, the point is determined as being too close to an edge, and it is a assigned a large score.

3.3 Feeding Data to Unreal Engine

After the final position for each node's location was finally generated, node positions - (x, y, z) coordinates - were printed out to a CSV file for use in Unreal engine. In addition, due to the lack of support for easily creating lines that follow a great circle path in Unreal engine, lists of points representing edges were also passed into the CSV file. As a tradeoff between line fidelity and speed of rendering, it was decided that each edge would be defined with ten points (which explains why for some edges the lines may look a little disjointed).

3.4 Implementation in VR

3.4.1 Hardware

Unreal Engine was chosen as the renderer of choice because of its built-in VR tools. The Unreal Engine unfortunately does not fully support Oculus Remote controllers, due to the lack of their commercial ability. However, as we wanted to enable active interaction while viewing a visualization, we further pursued it. This required that we do some significant debugging and digging around the Oculus Source code, and we were ultimately able to set it up with the help of a friend at Oculus.

3.4.2 Syncing with Data Source

Unreal was also chosen because it comes with built-in database tools, though they are extremely limited in functionality. The internal CSV parser requires that attributes in the first column of each row are unique. As such, the only way to search for a specific row is with the unique identifier. In short, this was a significant technical roadblock. We were unable to find suitable plugins, and most of the ones we found were for older versions that do not support the motion controllers. This also required that we change the schema - leading to inefficient querying within the renderer.

3.4.3 Rendering

We were surprised that Unreal does not come with a built-in 3D line drawing tool. We tried numerous different methods as suggested by the internet. The first was connecting each node using particle beams; however this was too performance intensive and led to significant FPS drops even with our small dataset. The second was using Unreal's spline to draw and map B-splines (which would have also needed to be implemented and calculated in the python data processor) onto the sphere, but Unreal's spline tool does not support B-splines and would've required that custom construction of a plugin. Thus, the final implementation involved settled for using the debug line drawing tool, often used to debug ray tracing in Unreal. These worked well as these lines were easy to render realtime. However, these required that interpolation points be passed from the python position generator to the Unreal engine.

3.4.4 Interaction

At first, the entire project was planned to function in conjunction with Oculus Touch motion controllers, however we were unable to acquire permissions to use the Touch for our project. Thus, we used the Oculus Remote for basic interactions. In an attempt to design the visualization to be easily viewable, we implemented rotation into the visualization. Oculus remote directional buttons were used to rotate the graph, which required re-rendering the edges every 0.01 seconds. In addition, the user was also allowed the ability to view multivariate data associated with each node. By pressing the select button, the user can trace a ray from his/her camera to the first object hit - if the ray hits a datapoint, text is displayed by that point providing specific information about the selected node. One point of interaction we would have liked to implement was real time data filtering in VR, but due to Unreal's flimsy database support, a large number of hardware related bugs and lack of time, we decided against it.

4 Results and Further Work

The end result of the discussed work is a pipeline for converting raw network data into a 3-D interactive visualization designed for use with VR. The work has proved the viability of VR data visualization and shown a simple example of how it can be implemented. A sample of an implemented visualization is shown in Figure 3.

The currently finished work is encouraging evidence supporting the concept of data visualization in VR. However, there are many future avenues of work related to this concept that we hope to pursue. From a VR perspective, we believe that supporting real time in-visualization data filtering would be the next step. However, this would probably have to involve using a list of preset filters

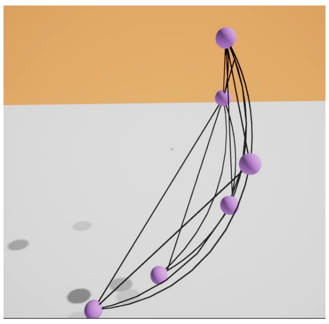

Figure 3: Sample View of Generated Visualization

- numerical or textual input in Unreal is currently unwieldy. Currently implementable solutions to this problem could include button filters or sliders. Other interaction that could be improved in VR could include using the controllers to perform more explicit actions such as grabbing data, changing camera location, and edge selection. Further optimization of the space-filling algorithm (a lesser priority of the full work) could involve modifying the cost function used in perturbing the variables such that it includes node-specific variables. For example, if nodes on a graph represent individuals in a social network, a variable such as their geographic location could be included in the cost function to encourage geographically close individuals to be close on the spherical mapping of the nodes. In any case, there are a lot of potential directions for continued research and implementation.

5 Conclusion

Data Visualization in virtual reality is still an emerging field, with its standards yet to be set. We wanted to use this opportunity to explore 1) the challenges of rendering graph visualizatoins in VR and 2) exploring interaction with data in VR. Because traditional 3D visualizations do not account for change of perspective, these techniques are suboptimal in virtual reality. This is particularly true in graph visualizations, when certain optimizations, such as minimizing edge-crossing, may be less important user-perspective is constantly changing. We used a method that maps graphs to a spherical surface, optimized for rendering in VR. Our implementation also allowed for basic interaction with the graph, such as rotation and selecting a particular node to view multivariate data.

6 References

References

[1] Bryson, Steve. "Virtual reality in scientific visualization." Communications of the ACM 39.5 (1996): 62-71.

[2] Cruz-Neira, Carolina, Daniel J. Sandin, and Thomas A. DeFanti. *"Surround-screen projection-based virtual reality: the design and implementation of the*

CAVE." Proceedings of the 20th annual conference on Computer graphics and interactive techniques. ACM, 1993.

[3] Kwon, Oh-Hyun, et al. *"Spherical layout and rendering methods for immersive graph visualization."* Visualization Symposium (PacificVis), 2015 IEEE Pacific. IEEE, 2015.

[4] Wu, Yingxin, and Masahiro Takatsuka. *"Visualizing multivariate network on the surface of a sphere."* Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation-Volume 60. Australian Computer Society, Inc., 2006.