# AN OPEN-SOURCE, HARDWARE ACCELERATED INDUSTRY PROTOTYPE FOR MULTIDIMENSIONAL GLYPHS

Dylan Rees
*Swansea University*
*Swansea, UK*

Robert S. Laramee
*University of Nottingham*
*Nottingham, UK*

**ABSTRACT**

Data-glyphs are a common, useful metaphor for visualizing multidimensional data. However, to represent a larger number of glyphs requires an increasing amount of computational power to derive the data-dependent geometry of each glyph. We present a novel visualization implementation for computing and rendering customized data-glyphs for an industry application. Our software utilizes the parallel processing power of the Graphics Processing Unit (GPU) to enable interactive exploration of thousands of multidimensional data-glyphs. This is achieved by passing normalized data values to the OpenGL geometry shader to construct the geometry of each glyph reflecting the data values. The technique is applicable to a variety of glyph designs and supports fast selection between different designs. We demonstrate this technique to render more than 100,000 glyphs at interactive frame rates. To address the challenge of reproducing the technique, we provide a simplified open-source code example implementation that can be used by anyone, including our industry partner.

**KEYWORDS**

Glyphs, OpenGL, Rendering.

## 1. INTRODUCTION

Glyphs provide a popular means for conveying multidimensional data; however, their shape and layout can be complex and therefore computationally intensive to calculate their geometry. This is especially true for glyphs where the geometry of the glyph is dependent on the data dimensions to be represented. With an increasing number of glyphs required for ever growing data-sets, this issue is amplified, requiring longer computational time to render a complete image. This becomes a challenge where glyphs are used for interactive exploration of data, with scenes requiring continuous update. Scenes that are slow to update can be frustrating for users, making adoption of a tool with such a feature difficult.

In this paper we provide technical details of our software implementation for rendering a large number of multi-dimensional glyphs such as seen in Figure 1 (left). This is done by leveraging the parallel processing capabilities of a Graphics Processing Unit (GPU).

The contributions of this paper are:

- A novel implementation for calculating and rendering large numbers of multidimensional data-glyphs
- A method to support easy and fast selection of glyph design
- The provision of a self-contained open-source code implementation to demonstrate the technique

## 2. RELATED WORK

A survey of data visualization books by Rees and Laramee (Rees & Laramee, 2019) highlight six OpenGL textbooks that introduce and detail the operation of OpenGL (Angle & Shreiner, 2011), (Wolff, 2011), (Movania, 2013), (Wright Jr, et al., 2015), (Kessenich, et al., 2016).

McNabb and Laramee (McNabb & Laramee, 2017) describe four surveys addressing glyph design (Ward, 2002), (Borgo, et al., 2013), (Fuchs, et al., 2017), (Ropinski, et al., 2011). The work of Ward presents a taxonomy of glyph placement strategies (Ward, 2002). Borgo et al. provide design guidelines for glyphs (Borgo, et al., 2013). The work of Fuchs et al. surveys glyph based user-studies (Fuchs, et al., 2017) and Ropinski et al. survey glyphs for medical visualization (Ropinski, et al., 2011). Consultation of these surveys fails to identify work that discusses rendering implementation techniques for glyphs.

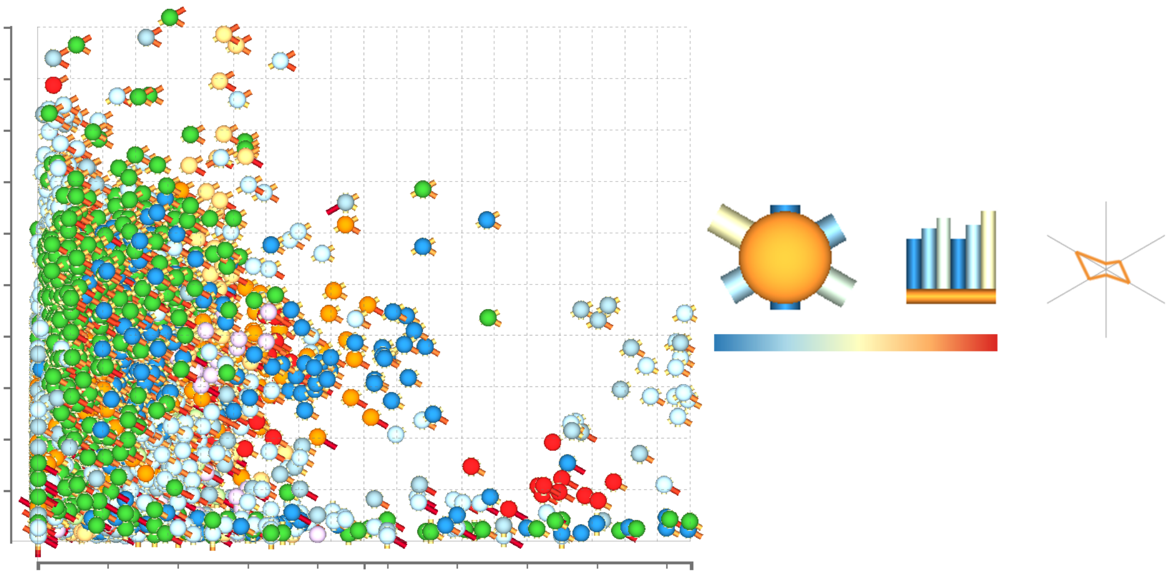
Figure 1: (Left) Over 6,500 glyphs drawn utilizing the technique presented in this paper. Each glyph features six continuous variables plus a categorical variable mapped to color. The x and y position of the glyph on the canvas are also mapped to variables. (Right) Three glyph designs, each of which feature six continuous variables represented by the size of the bars in the left two glyphs and by the radar on the right glyph. A categorical variable is also represented by color, indicated by the orange in these examples. Color is also used to redundantly map to the bar size on the right two glyphs, as indicated by the pictured colormap

The use of glyph optimization work is most prominent in scientific visualization, with glyphs often used to represent flow (Peng, et al., 2009), (Peng, et al., 2011) and for molecular visualization (Grottel, et al., 2014). Because of the large number of glyphs required to effectively visualize the flow or molecules, optimizations through the use of GPU rendering techniques are widely developed (Crawfis & Max, 1993), (Reina & Ertl, 2005), (Grottel, et al., 2009), (Peeters, et al., 2009), (Alharbi, et al., 2019).

Falck et al. compile and provide detailed explanations of GPU accelerated algorithms for rendering and processing scientific data (Falk, et al., 2016). As such, we recommend readers consult this work for a comprehensive overview of GPU accelerated glyph rendering.

Our work differs from others as we utilize GPU shaders for the construction of multi-dimensional data glyphs for information visualization rather than scientific visualization. In contrast to previous work, the technique we present leverages the geometry shader to construct each individual data-glyph geometry. We also provide a seldom-seen open-source implementation.

## 3. GLYPH RENDERING

In this section, we discuss step-by-step the process for replicating the technique that we present in this paper. The technique treats each glyph as a point primitive and passes the data variables from a vertex buffer through the vertex shader to the geometry shader. Once in the geometry shader, data values are used to determine the complex geometry of each glyph before being passed on in the pipeline to be rendered. This technique enables for complex glyphs to be rendered using a single draw call.

Before glyphs can be rendered, data must first be processed into a format that can be loaded into vertex buffers for rendering. The first step is to find the maximum and minimum value for each of the variables. Once this has been established, the next step is to normalize each of the variables to a range [0,1].

The data is then ready to be stored into vertex buffers to be transferred to the GPU memory. Before this, the number of variables each glyph is to represent must be established a priori. We demonstrate glyph designs in Figure 1 (right), inspired by the work of Rees et al. (Rees, et al., 2020), featuring six continuous variables plus a categorical value mapped to color. As well as the glyph variables, we also map the position of the glyphs to a two-dimension scatterplot enabling each glyph to represent an additional two variables. For the construction of these glyphs therefore requires nine data variables to be copied into vertex buffers and onto the GPU memory to be processed by the rendering pipeline.

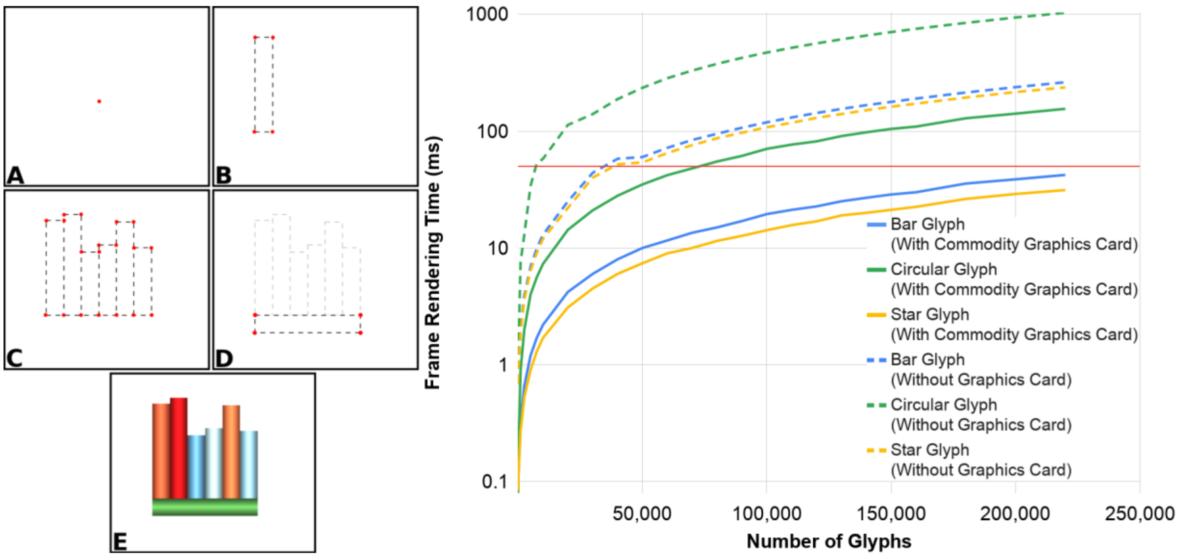

Figure 2: (Left) OpenGL shader glyph construction. A shows the single vertex position output from the vertex shader. B shows the output of a single bar from lines 42-57 of the geometry shader, while C shows the output of all bars in the encapsulating for loop, lines 28-58. The category indicator bar shown in D is output from lines 60-75 from the geometry shader. Finally, color is added in E by the fragment shader (see supplementary material). (Right) Rendering performance against the number of glyphs drawn for three different glyphs with and without a commodity graphics card

The vertex shader determines the positioning of the glyph, and simply passes the other data through to the geometry shader.

The geometry shader is used to determine the shape of the glyph according to the data variables passed from the vertex shader. The geometry shader for each of the glyphs in Figure 1 (right) can be found in the supplementary material or on the online repository (Anon, 2020).

The geometry shader takes in the point vertex position from the vertex shader and outputs a series of vertices in a triangle strip to build a bar chart glyph, this is demonstrated in Figure 2 (left). First the shader calculates the position in the screen-space, then iterates through each data variable to calculate the size of each corresponding bar, emitting five vertices for each bar to the fragment shader using the `EmitVertex()` function. Details being emitted are the positions for each new vertex, a color attribute to map the bar color to the data value, a 2D vector for shading effects, and an attribute to distinguish the data bars from the categorical identifying bar. Once the data bars have been emitted, vertices for an additional categorical identifying bar are emitted.

A variety of different geometry shaders, creating different styles of glyphs, can be used and easily switched by re-running the OpenGL program with the different shader, without the need to reload the data into the graphics memory buffers.

Once a vertex has been emitted by the geometry shader, it is processed by the fragment shader. The fragment shader adds shading effects and color to the vertices emitted from the geometry shader. A texture can also be added to the glyphs, therefore, if a glyph design incorporates an identifying symbol, this can be added as a texture at this stage of the process.

A self-contained, simplified open source implementation of the code is available in the supplementary material and on GitHub:

https://github.com/glyph-renderer/glyph

The open-source implementation developed for this article features three individual glyph designs, however additional user specified glyphs can be added with the addition of a new geometry shader.

## 4. DISCUSSION

A limitation exists in the amount of data that can be passed between shaders. This is dependent on the graphics hardware, for our consumer grade Nvidia GeForce GTX 1060 6 GB graphics card this is 1024 floating points of data. The number of glyphs that can be interactively draw utilizing this method is only limited by the hardware capabilities. However, with the games industry driving GPU development, this limitation becomes less relevant as new GPUs increase capabilities.

Drawing so many glyphs highlights an overplotting issue where the glyphs are drawn on top of one another become indistinguishable, as can be seen in Figure 1 (left). Because the technique introduced in this paper allows for interactive frame rates, a user can interactively zoom and explore the glyphs, partially alleviating this issue. Another solution to this is to cluster overlapping glyphs as previously presented by several works (Yang, et al., 2003), (Fuchs, et al., 2016), (Rees, et al., 2020).

To gauge the performance of the presented technique, the rendering time for each frame was measured for a varying number of glyphs. Results are shown in the supplementary material. Figure 2 (right) shows the performance of the open-source implementation presented in this paper. Performance is measured on two separate personal computers (PC). The first PC features an Intel i7-6700HQ processor, 8GB of RAM, and an NVIDIA GeForce GTX 1060 6 GB graphics card. The second PC features an Intel i5-6500 processor, 16GB of RAM, and has no dedicated graphics card, relying on integrated graphics (Intel HD Graphics 530). For compatibility, testing was performed using OpenGL version 4.5. Performance is measured for three glyph types, a bar chart glyph, a circular glyph, and a star chart glyph all depicted in Figure 1 (right).

Due to a limitation on array sizes, the maximum number of glyphs rendered is limited to 220,000. The PC including the graphics card achieved higher frame rates in comparison to the PC without for the same number of glyphs. Due to more calculations being required to compute the geometry, the circular glyph is slower than the bar chart glyph. In all scenarios, a frame rendering time of less than 17 ms was achieved for up to 2,500 glyphs (equivalent to 60 fps) and 50 ms (20 fps) for up to 7,500 glyphs. Over 50 ms the user will notice severe roughness. The PC without a graphics card takes 35 ms to render a frame with 5,000 circular glyphs, and 58 ms to render 10,000 circular glyphs but remains under the 17 ms threshold for 10,000 bar chart and star glyphs. The 50 ms frame rendering threshold is only crossed for 40,000 bar chart or star glyphs.

The PC with the graphics card remains under a 17 ms frame rendering time for over 20,000 glyphs, with over 70,000 circular glyphs rendered before the frame rendering time increases over the 50 ms threshold. For the bar chart glyphs, the PC with the graphics card remains under 17ms for 90,000 glyphs and under 40 ms for 200,000 glyphs. Over 120,000 star glyphs are rendered before the frame rendering time increases over 17ms and remains under 30ms for 200,000 star glyphs. With every glyph design tested, both with and without a graphics card, overplotting becomes a greater issue before frame rates drop below an interactive rate.

## 5. CONCLUSION

We describe a technique for generating and interactively rendering many unique multidimensional data glyphs based on instance variables. This technique utilizes GPU parallelization to calculate the individual geometries of each glyph instance. Source code for a simple version of the technique presented are provided to enable ease of recreation. In the future we aim to performing an exploratory analysis of the limits of the presented technique, gauging the maximum number of glyphs that can be rendered while still achieving interactive frame rates (5 fps). A solution for overplotting previously mentioned is another future work aim.

## REFERENCES

Alharbi, N. et al., 2019. Hybrid Visualization of Protein-Lipid and Protein-Protein Interaction. *Proceedings of Eurographics Workshop on Visual Computing in Biology and Medicine (VCBM).* Brno, Czech Republic.

Angle, E. & Shreiner, D., 2011. *Interactive Computer Graphics: A Topdown Approach with Shader-Based OpenGL.* Addison-Wesley, Boston USA.

Anon, 2020. *glyphRenderer – GitHub* https://github.com/glyph-renderer/glyph Accessed Apr 27, 2020.

Borgo, R. et al., 2013. Glyph-based Visualization: Foundations, Design Guidelines, Techniques and Applications. *Proceedings of Eurographics 2013 - State of the Art Reports*. Girona, Spain, pp. 39–63.

Crawfis, R. A. & Max, N., 1993. Texture splats for 3D scalar and vector field visualization. *Proceedings of Visualization'93*. San Jose, USA, pp. 261–266.

Falk, M. et al., 2016. Interactive GPU-based Visualization of Large Dynamic Particle Data. *Synthesis Lectures on Visualization,* Vol 4, pp. 1–121.

Fuchs, J. et al., 2017. A systematic review of experimental studies on data glyphs. *IEEE Transactions on Visualization and Computer Graphics,* Vol 23, pp. 1863–1879.

Fuchs, J. et al., 2016. Leaf Glyphs: Story Telling and Data Analysis Using Environmental Data Glyph Metaphors. *Proceedings of Computer Vision, Imaging and Computer Graphics Theory and Applications.* Berlin, Germany, p. 123–143.

Grottel, S. et al., 2014. MegaMol—a prototyping framework for particle-based visualization. *IEEE transactions on visualization and computer graphics,* Vol 21, p. 201–214.

Grottel, S. et al., 2009. Optimized data transfer for time-dependent, GPU-based glyphs. *Proceedings of 2009 IEEE Pacific Visualization Symposium*. Beijing, China, pp. 65–72.

Kessenich, J. et al., 2016. *OpenGL Programming Guide: The Official Guide to Learning OpenGL.* Addison-Wesley Professional, Boston, USA.

McNabb, L. & Laramee, R. S., 2017. Survey of Surveys (SoS) - Mapping The Landscape of Survey Papers in Information Visualization. *Computer Graphics Forum,* Vol 36, pp. 589-617.

Movania, M. M., 2013. *OpenGL Development Cookbook.* Packt Publishing Ltd., Birmingham, UK.

Peeters, T. H. J. M. et al., 2009. Fast and sleek glyph rendering for interactive HARDI data exploration. *Proceedings of IEEE Pacific Visualization Symposium*. Beijing, China, pp. 153–160.

Peng, Z. et al., 2011. Mesh-driven vector field clustering and visualization: An image-based approach. *IEEE Transactions on Visualization and Computer Graphics,* Vol 18, pp. 283–298.

Peng, Z. et al., 2009. Glyph and Streamline Placement Algorithms for CFD Simulation Data. *Proceedings of NAFEMS World Congress (NWC) Conference,* Crete, Italy, pp. 66.

Rees, D. & Laramee, R. S., 2019. Survey of Information Visualization Books. *Computer Graphics Forum,* Vol 38, pp. 610-646.

Rees, D. et al., 2020. AgentVis: Visual anaysis of agent behaviour with hierarchical glyphs. *IEEE Transactions on Visualization and Computer Graphics.*

Reina, G. & Ertl, T., 2005. Hardware-accelerated glyphs for mono-and dipoles in molecular dynamics visualization. *Proceedings of EUROVIS 2005: Eurographics / IEEE VGTC Symposium on Visualization,* Leeds, UK, pp. 177–182.

Ropinski, T. et al., 2011. Survey of glyph-based visualization techniques for spatial multivariate medical data. *Computers & Graphics,* Vol 35, pp. 392–401.

Ward, M. O., 2002. A taxonomy of glyph placement strategies for multidimensional data visualization. *Information Visualization,* Volume 1, pp. 194–210.

Wolff, D., 2011. *OpenGL 4.0 Shading Language Cookbook.* Packt Publishing Ltd., Birmingham, UK.

Wright Jr, R. S. et al., 2015. *OpenGL SuperBible: Comprehensive Tutorial and Reference.* Addison-Wesley Professional, Boston, USA.

Yang, J. et al., 2003. Interactive hierarchical displays: a general framework for visualization and exploration of large multivariate data sets. *Computers & Graphics,* Vol 27, pp. 265-283.