Overplotting: Unified solutions under Abstract Rendering

Joseph Cottam and Andrew Lumsdaine
Indiana University
Center for Research in Extreme Scale Technologies (CREST)
Bloomington, IN, USA
{jcottam,lums}@indiana.edu

Peter Wang
Continuum Analytics
Austin, TX, USA
pwang@continuum.io

***Abstract*—It is impossible to directly visualize all of the items of a large dataset at once. Often, the number of items exceeds the number of pixels. Since direct representation is not a reliable option, a variety of methods have been developed for dealing with indirect representation. Such methods include clustering and intelligent filtering to reduce the number of items being considered in the first place. However, these techniques impose a high computational and interpretation costs. The alternative is to employ techniques to directly deal with the over-plotting that occurs. that occurs when there are too many items to display without overlapping. Over-plotting techniques include alpha composition, color weaving and selective plotting. Each of these has variants that yield different cognitive or computational optimizations. Unfortunately, most advanced over-plotting techniques are wrapped up in specific libraries. Experimenting with different techniques is cumbersome because they have not been provided with uniform interfaces or in a single runtime. This paper presents Abstract Rendering, a recasting of the rendering process that enables concise expression of many over-plotting techniques. Furthermore, the Abstract Rendering formulation yields efficient execution strategies. Combined, it is practical to explore different over-plotting techniques for large data without requiring significant alteration to existing pipelines.**

***Keywords*-Rendering, Overplotting, High-Definition Alpha Composition**

I. Introduction

Visualization transforms source information into a set of pixels. The info-vis reference model (Figure 1) provides a vocabulary for discussing that transformation [1]. Information visualization frameworks tend to focus on the Visual Mappings stage, where raw data is projected into a set of geometrical abstractions, and graphics are represented with high precision on a logical canvas. Conversion to actual pixels is given significantly less attention. Many frameworks simply offload the view transforms and related rasterization

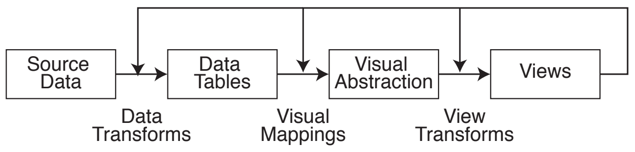


Figure 1. Information visualization reference model [1].

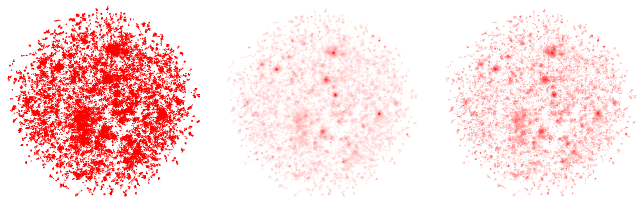


Figure 2. Cluster-based arrangements of Sourceforge.net social network nodes under various rendering conditions.

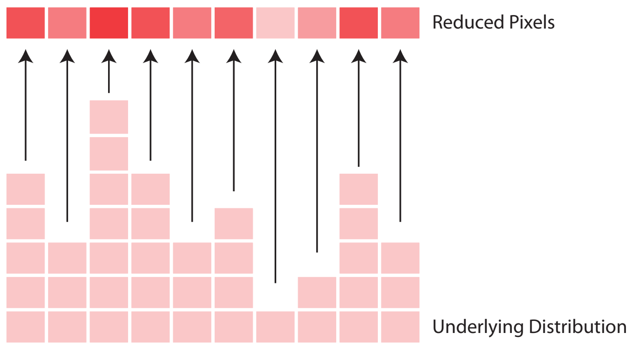


Figure 3. One-dimensional distribution and reduction.

to external graphics libraries (such as an SVG renderer, OpenGL, or Java2D). Efficient transfer to the underlying engine is often the only consideration discussed. Abstract Rendering expands control over what occurs in the View Transform stage of the info-vis reference model.This control enables direct and simple discussion of render-time effects pertinent to visualization construction.

Consider the examples in Figure 2. The plot contains 27,500 points, representing the social network formed at surrounding Sourceforge.net circa 2007. (A more complete description of the datasets and its treatment is found in Section II.) With each node allocated a three-by-three space, there are insufficient pixels to display this dataset in less than a 500 by 500 pixel region, regardless of layout. Since data layout is usually not a fully regular tiling of a space, treatment of even this modest dataset encounters overplotting. Figure 2a shows 100% opaque squares, giving the rough distribution of the clusters in the data but with significant over-plotting leading to occlusion. When over-plotting is present, the rendered plot conveys presence/absence distribution information, which is a bound on the actual distribution and not a full description. This is not necessarily undesirable, but it is a silent transition of plot type that is not represented in the visual mappings. The issue of silent transitions is compounded by the fact that the visual mappings are often presented as the definition of a visualization [2]–[6]. A common approach to mediate data loss due to over-plotting is to use alpha compositing, providing more complete distribution information on a per-pixel basis as an emergent effect (Figure 2b). In this minimal treatment, the highest peaks are still over-saturated and the lowest troughs are indistinguishable from empty. In many cases, this over-saturation is partially determined by hardware constraints and outside of the scope of control of traditional visualization frameworks. Taking into account the actual quantities in the most saturated pixel, Figure 2c accurately represents the whole dynamic range while persevering minimum visibility on the least-saturated pixel.

Abstract Rendering provides direct access to many over-plot-related techniques by exposing the rasterization and implicit binning process that occurs when rendering to pixels (see Figure 3). Succinctly, Abstract Rendering constructs a synthetic data space that is informed by both geometric data representations and render-related rasterization. It logically proceeds in two phases. The first phase is the creation of a synthetic data space between the geometric representations and the raw pixels of an image. The second phase is application of transformations to that synthetic space to create an image.

This first phase is analogous to traditional rendering but no restriction is placed on the topology or content of the rendering (where traditional rendering produces regular rectangular grids of colors). To be effective, the synthetic space must provide new affordances to analysis that neither the raw data, pixels, or geometric constructions do by themselves. The ability to compute the full range of overlap is one such affordance (discussed earlier with respect to Figure 2).

The fundamental observation is that individual pixels represent raw data by way of a synthetic space built out of (but not identical to) the geometric representation. Abstract Rendering (1) reifies this synthetic space and (2) provides tools for working with that space. The system is named Abstract Rendering because it computes a discretized result, much like standard rendering, but the discrete values do not need to be colors (and are thus more abstract than colors).

II. Abstract Rendering

Abstract rendering is done by chaining together four different function types. The four function types are (1) select, (2) info, (3) aggregate, and (4) transfer. The first three functions are used to create an initial synthetic data space. This synthetic data space is derived from the source data and the geometric data. The select function picks geometry, the info function processes individual geometric items, and the aggregate function combines info values together. An effective synthetic data space enables efficient analysis, performed by the transfer function. These four functions will be described in greater detail in turn through the evolution of an example visualization.

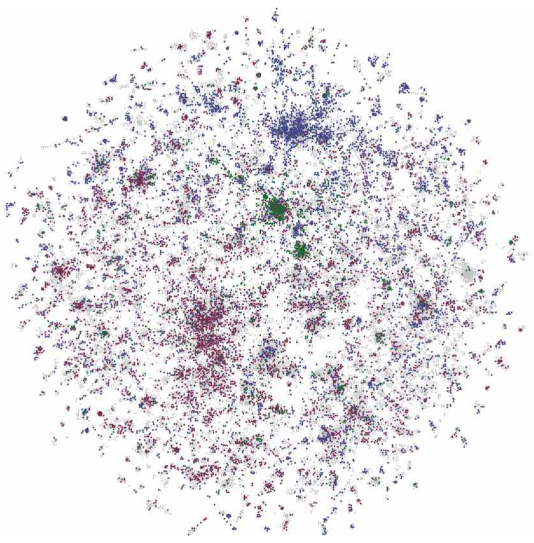


Figure 4. Categorical treatment of Sourceforge.net social network nodes.

A. Dataset

The example dataset comes from a social network analysis of the open source community surrounding Sourceforge.net circa 2007. Each project is a node in the network and each developer is a link. The largest connected component (27,500 projects) was laid out using VxOrd [7]. This layout was used to visually assess the interactions between the social network and various project attributes as part of an exploratory analysis [8].

Figure 2 shows various renderings of just the project nodes, encoded as red squares. The naive treatment of 100% opaque squares (Figure 2a) demonstrates the rough distribution of the clusters in the data. However, it does not show density inside of the clusters. Figure 2b shows a simple alpha-treatment (alpha set to 1%). In this minimal treatment, the highest peaks are still over-saturated and the lowest troughs are visually indistinguishable from empty. Shifting to 25% alpha on each node yields improved definition in the low-population areas, but at the cost of significant over-saturation in the highest peaks. Abstract Rendering enables an accurate treatment of this data.

B. Simple Example: Select, Aggregate, Transfer

As mentioned, the original dataset is being represented as colored squares. Abstract rendering accepts these squares as input. The basic task for an accurate transparency-based representation of the data is to count how many items land in each pixel, then to create a color ramp that handles the entire range of counts. This pixel-level analysis and scale creation is know as high-definition alpha composition (HDAlpha) [9].

For high-definition alpha composition, the synthetic data space is a pixel-level grid of the number of items that land on each pixel. This grid of counts is directly constructed by first selecting all the items that land on a pixel and, second, counting the number of items that are selected. In terms of the four functions in Abstract Rendering, the selector is the "intersects" function, which selects all of the items that intersect a given pixel. The aggregator is the "length" function that counts the number of items in a list. (The "info" function will be addressed later.) Applying $length(intersects(x, y, G))$ for each pixel in the image will result in the required synthetic data space. (Where x, y indicate a single pixel of the screen projected into the logical canvas that the glyphs occupy and "G" is the geometric representation of the source data.)

The synthetic data space can be directly analyzed to find the minimum and maximum intersection values. A color ramp can be built between those two values and directly applied to the synthetic data space. Building and applying this color ramp is the task of "transfer," the fourth Abstract Rendering function. $interpolate(s, Red.10, Red, min(S), max(S))$ therefore supplies the color for each pixel ("Red.10" is red with 10% alpha, S is the synthetic data space and s is an item in S). Applying the Abstract Rendering process yields the image shown in Figure 2c. This avoids over-saturation in the highest peaks, while guaranteeing visibility of the lowest troughs (shown at 10% alpha). The result is a correct image of the distribution of the nodes of the social network.

C. The Info Function

In the preceding example, the "info" function was not specified because it was not required. The count of the values was the point of concern. Trivially, the same encoding can be created by using an info function of id that returns whatever it is passed (i.e., $id(x) = x \forall x$). Using id for the info function, the synthetic space creation is done by $length(id(g)|g \in intersects(x, y, G))$. Recalling the original problem, each Sourceforge project has various attributes associated with it. The "info" function slot enables a more detailed representation that includes those attributes. Assuming the attribute is stored with the geometry in G and encodes the programing language with the values "Python", "C/C++", "Java," and "Other", a more detailed synthetic data space can be made with $countCategories(att(g)|g \in intersects(x, y, G)))$. In this phrasing, "att" returns the attribute value and "countCategories" creates a list of the unique values seen paired with how often they are seen. Applying a category-aware transfer function, the image in Figure 4 results.

D. Formalization

Abstract Rendering can be compactly described as the application of four functions, combined in the following fashion:

$$s_{xy} = Aggregate(\{Info(g)|g \in Select(x, y, G)\})$$
$$c_{xy} = Transfer(s_{xy})$$

In this formulation all (x,y) values refer to positions on the screen and must match up between the two equations to color a single image pixel. Furthermore, G represents all glyphs with $g \in G$, s_{xy} represents a value in the synthetic data space and $c_{x,y}$ is a final pixel color. Each of the component functions may require additional arguments (such as "interpolate" needing the high color, low color, min value and max value). When required in further discussion, these will be included in line. In general, the first of these equations will be referred to as the synthesis step, while the second will be called the transfer step.

The final Sourceforge image as seen in Figure 4 is produced with the following abstract rendering equations:

$$\begin{aligned} s_{xy} &= countCategories(\{att(g) \\ &\qquad |g \in intersects(x, y, G)\}) \\ s'_{xy} &= ReKey(s_{xy}, \\ &\quad \{Python : Green, C/C++ : Red, Java : Blue, ...\}) \\ c_{xy} &= HDAlpha(s'_{xy}, S') \end{aligned}$$

Where $ReKey$ replaces the keys of a dictionary with the keys found in a new dictionary and where $HDAlpha$ expects a set of colors as its first argument and a set that includes the most extreme values that will be encountered as its second. As seen above, sometimes it is convenient to construct multiple synthetic data spaces. In such cases, there are multiple instances of the transfer step, but the overall concepts remain the same.

To simplify further discussion, a few auxiliary concepts need to be defined. By convention, sets of items will be denoted by upper-case letters while members of a set will be denoted by the same lower-case letter and a subscript. (so s_{xy} indicates a member S). Related, any x/y value will refer to a pixel position. Some abstract rendering systems involve complex functions. These will be constructed via function composition. To facilitate composition of functions that take multiple arguments, this paper employs partial application, indicated by square-braces. Arguments to be supplied later are filled with $\diamond$. When partial application is used, a new function is produced with the same number of arguments as holes. For example, for function $F(a, b, c)$, the statement $F[1, \diamond, 3]$ yields a new function $F'(b)$. $F'(2)$ executes exactly as $F(1, 2, 3)$. To preserve generality, this

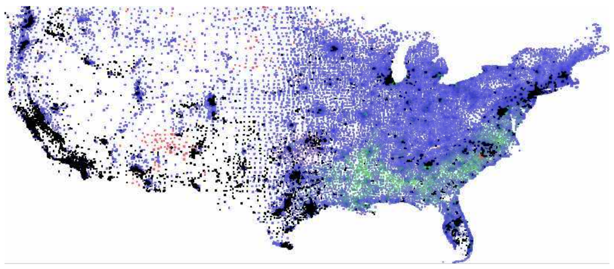


Figure 5. US Census tracts reporting at least 10% reporting "other" selected and plotted pure black.

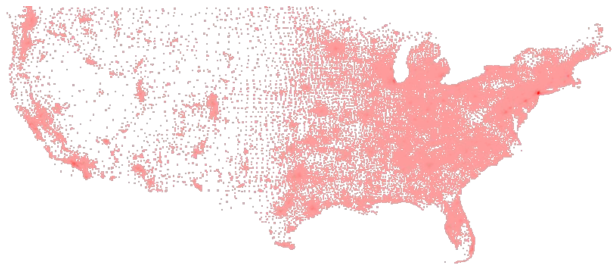


Figure 6. US Census tracts with all race information combined to form a net population map.

paper assumes that partial application occurs on a per-pixel basis (i.e., once per s_{xy}). This simplifies the definition of some functions.

III. Example Abstract Rendering Encodings

The general Abstract Rendering form can be used to describe many different techniques for overplotting. This section provides the equations for several existing techniques. Example figures provided in this section are derived from tract-level population and race data found in the 2010 US Census [10]. Unless noted, blue represents Caucasian descent, green represents African American descent, red for Native American descent and grey for all others.

A. Overplotting

$$\begin{aligned} s_{xy} &= color \circ max['Z, \diamond](\{id(g) \\ &\qquad |g \in intersects(x, y, G)\}) \\ c_{xy} &= id(s_{xy}) \end{aligned} \qquad \text{(Overplot)}$$

Basic overplotting occurs when pixels colors are assigned on a last-write-wins basis. To achieve overplotting in the Abstract Rendering framework, an ordering basis must be established. In the equation shown above, the ordering basis is the Z-value of the glyphs. The information function selects the color and the Z from each glyph, the reduction picks the color of the glyph with the largest Z value. The resulting aggregate-set is a list of colors. This equation is used for Figures 2a and 8a.

B. Selection-set Rendering

$$\begin{aligned} s_{xy} &= RLE \circ Sort['sel, \diamond](\{Id(g) \\ &\qquad |g \in intersects(x, y, G)\}) \\ c_{xy} &= SubstOn[\diamond, 'Sel, red, 'Color](s_{xy}) \end{aligned} \qquad \text{(Selection)}$$

While simple overplotting relies on a pre-arranged ordering of items, a more complex form of overplotting may use a function to decide which items to keep on top. In a traditional framework, this effect can be achieved by setting the z-order attributes. However, in Abstract Rendering it can be done in the final stages of the rendering pipeline. A common case for this is type of rendering prioritization is to keep selected values visible, as is done in Figure 5. Equation Selection realizes this type of prioritization, provided the glyph-set has a sel field that indicates which entries are selected. The RLE function computes a "Run Length Encoding", changing a list of values into a list of categories and the number of times they occur and in the order that they occur. In the most general case RLE, the list of values can have repetitions (just not consecutively). However, since RLE is composed with $sort$ in Equation Selection, the result is a list of unique values and their respective counts. Counts are sorted with respect to the sel field. This phrasing also relies on a $SubstOn(switch, val, alt, tuple)$ which is a wrapper around an if. If the field indicated by $switch$ is true, then val is returned, otherwise the value of the field in the tuple indicated by alt. A lookup function to a separately maintained selection index could be used in place of the Sel field.

C. Uniqueness in Neighborhood

$$\begin{aligned} s_{xy} &= color(\{min[count] \circ RLE \circ sort(g) \\ &\qquad |g \in neighbors(x, y, G)\}) \\ c_{xy} &= id(s_{xy}) \end{aligned} \qquad \text{(Unique)}$$

Selection set rendering applies a context-oblivious prioritization function. The items being of a particular pixel are all that are required to determine how to actually render the pixel. Equation Unique shows how to pull the most unique item to the front on each pixel. This type of screen-space calculation is very difficult to encode using more standard Visual Encodings since the visual encoding is directly influenced by the view transform.

D. Homogeneous Alpha Compositing

$$\begin{aligned} s_{xy} &= count(\{id(g)|g \in intersects(x, y, G)\}) \\ c_{xy} &= interpolate[low, high, S, \diamond](s_{xy}) \end{aligned} \qquad \text{(HomoAlpha)}$$

Instead of *selecting* which item to render on a particular pixel, blending the items of a pixel is a common technique. Alpha composition is the most common expression of blending.

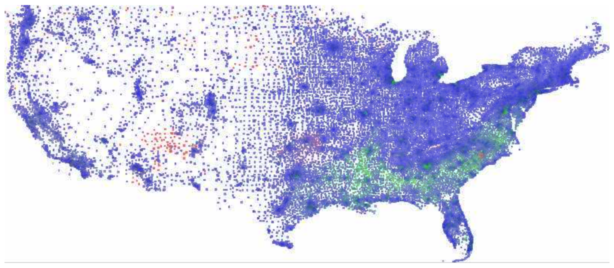


Figure 7. US Census Map with race data stratified and then composed.

Homogenous alpha composition occurs whenever all rendered items have the same visual representation. It is used to produce density representations (like those in Figures 2c and 6). The synthetics-set is made by counting the glyphs that intersect each pixel (using the $count$ function). The $interpolate$ function then interpolates from the low value to the $high$ value. The full synthetics-set (S) is used to establish the low and high input values. Working with the counts directly enables more fine-grained control over the interpolation, avoiding issues of unknowingly oversaturating the alpha-buffer. Therefore, the Abstract Rendering representation is trivially able to implement high-definition alpha composition [9].

E. High-definition Alpha Compositing

$$\begin{aligned} s_{xy} &= RLE['Z, \diamond](\{sort['Z, \diamond](g) \\ &\qquad |g \in intersects(x, y, G)\}) \\ c_{xy} &= \alpha compose \circ scale[S](s_{xy}) \qquad \text{(HighAlpha)} \end{aligned}$$

Full high-definition alpha composition extends the concern of buffer over-saturation seen in homogenous alpha composition to both the alpha and the color buffers. As with homogenous alpha composition, the key is to measure the extrema before apply the interpolation function. In Equation HighAlpha, the $\alpha compose$ function implements standard alpha composition [11]. The $scale$ function is a placeholder for the chain of functions that does the range measurements and interpolation for eventual composition in $\alpha compose$. Details on these range calculations and scalings can be found in earlier work [9], [12]. The necessary range information is derived from the synthetic-set (S). Equation HighAlpha equation is applied in Figures 7 and 8b.

High-definition alpha composition can be modified by sorting on the output color or a data field to achieve 'stratified' alpha composition. This stratification can be used to emphasize particular values in the plot (since alpha composition is order dependent) or provide more efficient rendering (for example, WebGL often performs faster when fewer pen-color changes).

F. Color Weaving

$$\begin{aligned} s_{xy} &= RLE(\{Sort \circ Color(g)|g \in intersects(x, y, G)\}) \\ c_{xy} &= Weave[S, x, y, \diamond](s_{xy}) \qquad \text{(Weave)} \end{aligned}$$

Color weaving takes an alternative tack to representing more than one item in a space than alpha composition. Rather than blending colors in one pixel, it represents a mosaic pattern of the original source colors throughout the space. The crux of the Abstract Rendering encoding shown in Equation Weave is the $Weave$ function, which is directly derived from the definition in Haleh, et al [13]. Using the two-phases of Abstract Rendering more completely, an alternative weaving implementation would defer actual color selection until transfer. In this alternative implementation, the categories would be woven instead of the colors themselves.

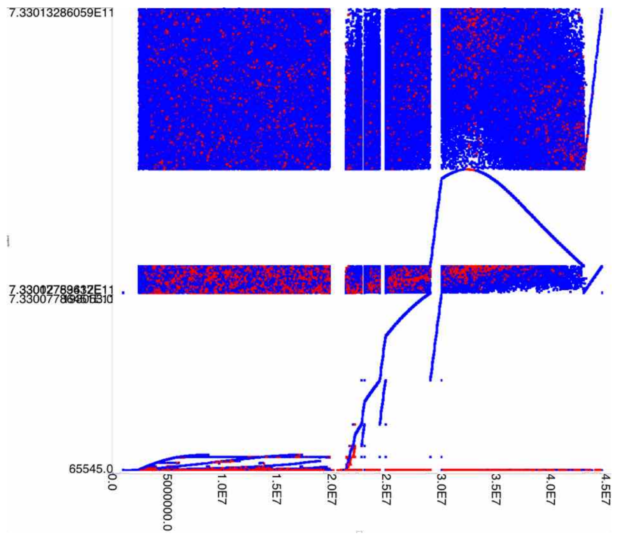
IV. Applications

The social network at Sourceforge and the USCensus were used earlier as set examples. However, neither represented particularly large dataset. This section provides two additional applications with larger datasets.

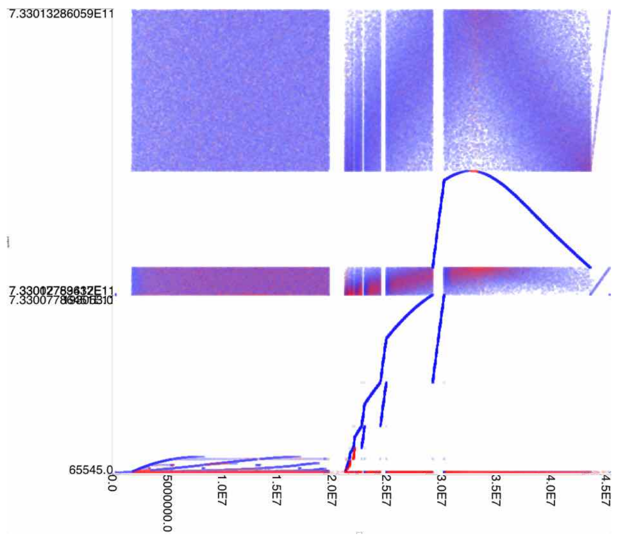
A. Memory Access Patterns

Abstract Rendering framework was applied to visualizing memory access in the Boost Graph Library (BGL). The BGL is a template-based C++ library for graph analysis. In graph analysis problems, memory access patterns are driven by the data present, and thus not amenable to many standard memory access optimizations. This is in contrast to matrix problems where accesses tend to be statically analyzable. In fact, mediated by the several layers of abstraction in the BGL, the actual access patterns were not well understood. In an attempt to further optimize the BGL, access patterns were recorded and analyzed. This section presents a portion of that analysis. (Preliminary analysis made it possible to determine where the program and various auxiliary data structures resided. These considerations have been omitted from this discussion.)

Figure 8 presents two treatments of the memory access data. In both cases, a single memory access is depicted as a point on the screen. There are 930,000 memory accesses presented. If the memory value was already in L2 or higher cache, the point is colored blue (a cache hit), otherwise it is colored red (a cache miss). Figure 8a is a naive projection that accepts the default iteration ordering via overplotting (see Section III-A) . In contrast, Figure 8b shows a full high-definition alpha treatment of the data (see Section III-E). This second image displays the subtlety of the memory access patterns, showing that a mixture of hits and misses is common but making some areas of near 100% hit or miss clear. Examining these two images, and other renderings based on percentage thresholds, helped the analysts understand the memory access patterns.



(a) Overplot



(b) High-Definition Alpha

Figure 8. Extreme treatments of BGL memory accesses patterns.

An important benefit of Abstract Rendering is the ability to change the transfer function but re-use the results of the synthesis step. On a Macbook Air, switching between transfer functions that do not consider the neighborhood required an average of 43ms for 500 by 500 pixel images and scaled with the number of pixels to 120ms for 1500 by 1500 pixel images (sizes tested at increments of 100 pixels and average of 10 executions). This enabled rapid comparison between many different treatments. Because transfer functions work on the results of the synthesis, not on the original input, the response level was independent of the input data size.

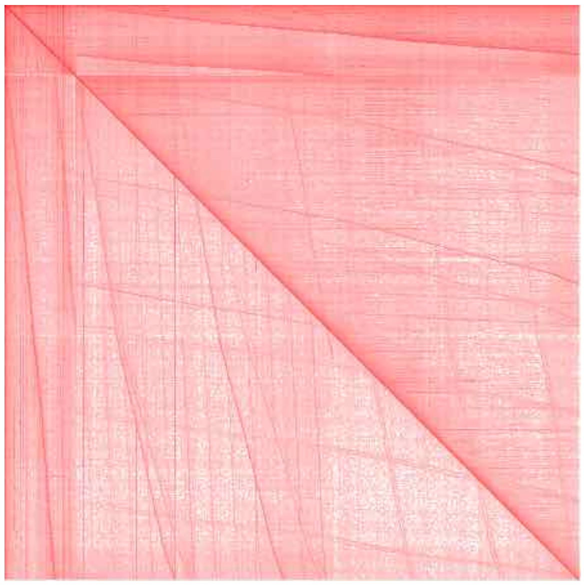


Figure 9. Adjacency matrix of the Kiva dataset with density log transformed.

B. Network Adjacency

The second dataset is a collection of 37 million transactions from the Kiva micro-finance site. Each transaction represents a monetary transfer between a lender, borrower, or intermediary. Each actor was given an identifier, with senders placed on the x-axis and receivers placed on the y-axis. Each transaction is represented as a point at the intersection of the sender and receiver. The coloring was done by log-transforming the counts of items contained in each pixel. The final visualization is shown in Figure 9 and the Abstract Rendering treatment is based on the one given in Section III-D.

As with the memory visualization presented earlier, the final visualization was not the only option explored. Significant overplotting prevents standard alpha techniques from being effective. At a 10% alpha level, over 50% of the screen is over-saturated. A linear transform is also ineffective because of an extremely skewed in the distribution (half of the transactions appear in the upper-left quadrant). Changing between the different transfer functions remained as responsive as the BGL examples, despite over 40 times more data in the Kiva dataset than in the BGL memory dataset.

For both the BGL and Kiva datasets, Abstract Rendering enabled rapid exploration of multiple representation techniques. In both cases, several encodings not presented here were constructed and compared. Construction typically consists of instantiating the relevant functional units from the library, typically just four lines of code (one for each function category). Comparison the proceeded through a simple utility that allowed the pre-coded combinations to be selected interactively. This breadth of exploration was not possible under other visualization frameworks because they lacked the ability to succinctly express the differences between the different overplotting treatments.

V. Implementation

The ability to interactively switch between different Abstract Rendering encodings is a signifiant advantage of Abstract Rendering over other libraries. Efficiency changing rest directly on implementation decisions.

Abstract Rendering has been implemented in Java and Python. This section describes the implementation in Java. The Python implementation is similarly structured, but differs in the parallelization strategy (relying heavily on vectorization through NumPy [14]).

The Abstract Rendering implementation follows the definition provided in Section II. Aggregate, info, and transfer functions are directly represented, though aggregate functions are slightly modified to provide efficient execution in out-of-core environments.

The select function is replaced with a "Renderer" class that controls the overall data access order. This includes access not just to the underlying dataset (as the select function implies) but also to the aggregate values (i.e., the order and frequency that the x/y values appear in). Renderers fall into two general categories: by pixel and by glyph. By-pixel renderers perform selection essentially as described in Section II. This strategy is efficient when used with datasets that are spatially arranged (such as a quad-tree). Any given glyph will be accessed once for each pixel the glyph touches. Many data structures do not efficiently handle the highly spatial nature of these queries and thus a pixel-oriented rendering strategy is not effective.

A glyph-based rendering strategy takes the opposite tack. Each glyph is visited exactly once, and each pixel may be updated multiple times. This enables efficient rendering when using non-spatial data structures because each glyph is visited in an order convenient to its container. However, the different iteration order means that synthetic values (i.e., the s_{xy} values) are typically updated multiple times. The aggregation function does not receive a list of info function results (as described in Section II). Instead it receives one info result and a pre-existing aggregate value. To compensate for this difference, aggregate functions must provide a "zero" value and must be commutative/associative (if deterministic execution is desired). The "zero" is used to initialize the synthetic data space. (When presented with a list of info results, the operator receives all of its operands "at once" and thus commutativity and associativity are not significant.)

The implementation supports a number of parallelization strategies and data configurations. A full analysis is beyond the scope of this paper. In brief, thread-level parallelism and vector-based parallelism have been explored. GPU, distributed memory and streaming data configurations have also been investigated. All use the components described above, augmented with various helper functions to facilitate data access or partial result combination. The out-of-core configuration is used in performance analysis (Section VI) and to handle the Kiva data set (Section IV-B).

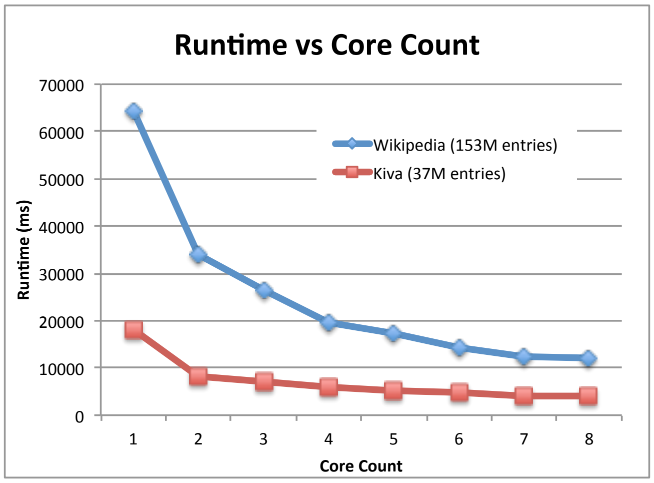


Figure 10. Scaling behavior of Abstract Rendering as processor count increases. Scaling behavior is near linear as processors as are added, but shows no additional improvements with hyper-thread processors (processors 8-15 are hyperthreads and improve by less than 5% vs. 8 cores).

VI. Performance

Section III described several Abstract Rendering phrasings, demonstrating its expressive capabilities. To be practical, the framework must be performant as well as expressive. An simple characterization of runtime performance was done with an eight physical core machine. Two adjacency matrix visualizations were used. The first was the Kiva dataset described Section IV-B. The second dataset is an adjacency-list of links found on Wikipedia, receiving the same treatment. The Wikipedia data set includes 153 million edges, representing the links of the largest connected cluster if the category system is ignored. Both data sets were binary-encoded adjacency-lists stored in a memory mapped file. The file contents were streamed off disk and rendered in a glyph-parallel strategy. These datasets were used because the data volume is sufficient to require out-of-core processing but require simple analysis to create a visual representation.

Figure 10 presents the average performance over 10 executions while keeping the core count fixed. In general, more processors are more helpful, but hyperthreading is not. The scaling characteristic is similar between the two datasets, but the difference shows the overhead of abstract rendering in general. Even though this Wikipedia data set is four times larger than the Kiva data set, the overall runtime is only three times longer, on average.

Overall, the performance numbers are generally supportive of interactive visualization applications.

VII. Future Work

Current Abstract Rendering implementations closely tie the bin-elements with the display resolution and region. This decision introduces view-dependent effects. View-dependent effects are used advantageously in high-definition alpha composition, but may not always be desirable. Developing techniques for avoiding these effects and guidelines for their usage is an ongoing effort.

Section V described implementation considerations. There are several unexplored options that may lead to more efficient implementations, or to implementations that run in more complex runtime environments. Options include distributed memory or efficient GPU execution. The idea of binning is shared inMens [15].

Abstract Rendering can be applied to more than just overplotting. High-definition alpha composition rests on the idea of measuring pixel-level information. This same idea can be applied to screen-space metrics for visualization evaluation [16], [17]. Such applications are also being explored.

The idea of the transfer function comes from scientific visualization. However, years of research into transfer functions has yielded many interesting techniques. Mixed rendering styles, and context-aware highlighting are strong candidates for exploration. Additionally, as noted in Section I, Z-ordering creates an implicit volume-like space. Some volume-based techniques from scientific visualization maybe more directly applicable by more literally applying this metaphor.

VIII. Conclusions

Visualizing large data sets inevitably runs into overplotting issues. By considering the rendering process as binning, Abstract Rendering provides a means to unify many overplotting techniques. Furthermore, those techniques can be encoded succinctly at compile time and executed efficiently at runtime.

References

[1] S. K. Card, J. Mackinlay, and B. Shneiderman, *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufman, 1999.

[2] M. Bostock and J. Heer, "Protovis: A graphical toolkit for visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1121–1128, 2009.

[3] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-Driven Documents," *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011. [Online]. Available: http://vis.stanford.edu/papers/d3

[4] J. A. Cottam, "Design and implementation of a stream-based visualization language," Ph.D. dissertation, Indiana University, 2011.

[5] L. Wilkinson, *The Grammar of Graphics*, 2nd ed. New York: Springer-Verlag, 2005.

[6] H. Wickham, "A layered grammar of graphics," *Journal of Computational and Graphical Statistics*, vol. 19, no. 1, pp. 3–28, March 2010.

[7] G. S. Davidson, B. Hendrickson, D. K. Johnson, C. E. Meyers, and B. N. Wylie, "Knowledge mining with VxInsight: Discovery through interaction," *Journal of Intelligent Information Systems*, vol. 11, no. 3, pp. 259–285, 1998.

[8] J. A. Cottam and A. Lumsdaine, "Extended assortitivity and the structure in the open source development community," in *International Sunbelt Social Network Conference*. International Network for Social Network Analysis, January 2008. [Online]. Available: http://www.insna.org/PDF/Awards/awards_ms_2007.pdf

[9] C. Muelder, F. Gygi, and K.-L. Ma, "Visual analysis of inter-process communication for large-scale parallel computing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1129–1136, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2009.196

[10] "National Historical Geographic Information System: Version 2.0," University of Minnesota, Minneapolis, MN, 2011. [Online]. Available: http://www.nhgis.org

[11] T. Porter and T. Duff, "Compositing digital images," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 253–259, Jan. 1984. [Online]. Available: http://doi.acm.org/10.1145/964965.808606

[12] J. Johansson, P. Ljung, M. Jern, and M. Cooper, "Revealing structure within clustered parallel coordinates displays," in *Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, ser. INFOVIS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 17–. [Online]. Available: http://dx.doi.org/10.1109/INFOVIS.2005.30

[13] H. Hagh-Shenas, V. Interrante, C. Healey, and S. Kim, "Weaving versus blending: a quantitative assessment of the information carrying capacities of two alternative methods for conveying multivariate data with color," in *Proceedings of the 3rd symposium on Applied perception in graphics and visualization*, ser. APGV '06. New York, NY, USA: ACM, 2006, pp. 164–164. [Online]. Available: http://doi.acm.org/10.1145/1140491.1140541

[14] T. E. Oliphant, *Guide to NumPy*, Provo, UT, Mar. 2006. [Online]. Available: http://www.tramy.us/

[15] Z. Liu, B. Jiang, and J. Heer, "immens: Real-time visual querying of big data," *Computer Graphics Forum (Proc. EuroVis)*, vol. 32, 2013. [Online]. Available: http://vis.stanford.edu/papers/immens

[16] J. W. Tukey and P. A. Tukey, "Computer Graphics and Exploratory Data Analysis: An Introduction," in *Proceedings of the Sixth Annual Conference and Exposition: Computer Graphics*. Fairfax, VA: Nat. Computer Graphics Association, 1985, pp. 773–785.

[17] A. Dasgupta and R. Kosara, "Pargnostics: Screen-space metrics for parallel coordinates," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1017–1026, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2010.184