

Towards Utilizing GPUs in Information Visualization: A Model and Implementation of Image-Space Operations

Bryan McDonnel, *Student Member, IEEE*, and Niklas Elmqvist, *Member, IEEE*

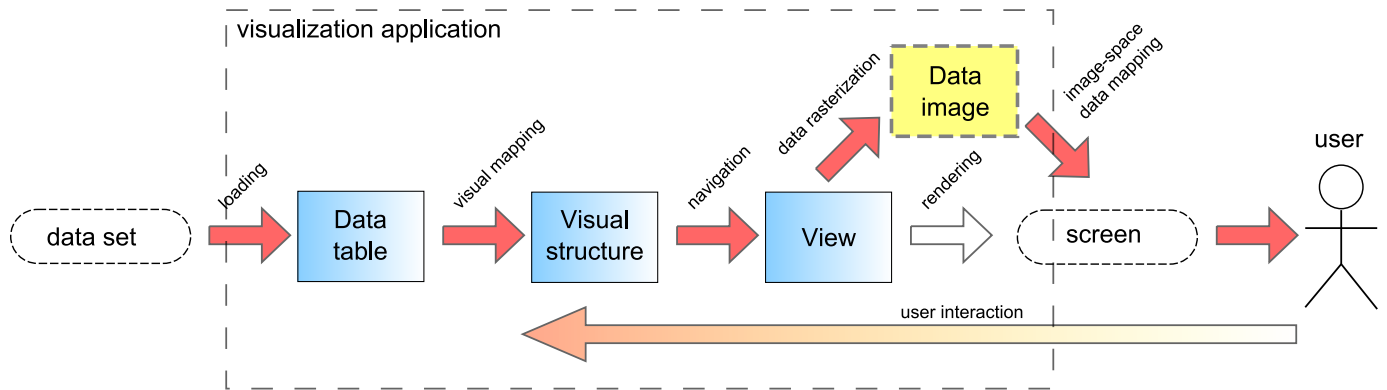


Fig. 1. The classic information visualization pipeline with transformations and states [6]. The “Data image” refinement is shown as a dashed state block. The final transformation is where the new image-space visualization operations are applied to the data buffer.

Abstract—Modern programmable GPUs represent a vast potential in terms of performance and visual flexibility for information visualization research, but surprisingly few applications even begin to utilize this potential. In this paper, we conjecture that this may be due to the mismatch between the high-level abstract data types commonly visualized in our field, and the low-level floating-point model supported by current GPU shader languages. To help remedy this situation, we present a refinement of the traditional information visualization pipeline that is amenable to implementation using GPU shaders. The refinement consists of a final image-space step in the pipeline where the multivariate data of the visualization is sampled in the resolution of the current view. To concretize the theoretical aspects of this work, we also present a visual programming environment for constructing visualization shaders using a simple drag-and-drop interface. Finally, we give some examples of the use of shaders for well-known visualization techniques.

Index Terms—GPU-acceleration, shader programming, interaction, high-performance visualization.

1 INTRODUCTION

Modern commodity graphics cards come equipped with their own processing units (GPUs) and are highly programmable, a recent change driven mainly by the computer games industry’s increasing needs for visual complexity and flexibility [36]. This new computing power is now also being used for general purpose computing [28]. Because the programmable graphics pipeline is based on 3D spatial concepts such as vertices and geometric primitives, the field of scientific visualization has been able to easily adopt GPUs for their own data [39]. However, the mismatch between these basic data types and more high-level and abstract datasets such as graphs, trees, and free text has meant that the sibling field of information visualization has been slow to catch on.

We present a refinement of the traditional information visualization pipeline [7] that extends the final stage of the pipeline with an image-space step and a set of image-space visualization operations (Figure 1). This provides a natural entry point for utilizing programmable GPUs even in information visualization. The benefit, beyond obvious performance improvements, is to support more flexibility in the display. For example, the method could be used for dynamic queries, adaptive color scale mapping, or data glyph rendering on the graphics card. In particular, the method also allows for offloading computations onto the

card, including operations such as correlation, filtering, and querying. The new image-space step also has the advantage of being conceptually consistent with existing visualization systems—it is present in all visualizations, although not all choose to expose it.

Exposing shader functionality in a visualization does not come without drawbacks, however: (1) the knowledge required to program GPU shaders poses an increasing gap between expert developers and visualization professionals [31], and (2) the shader languages themselves are poorly mapped to the visualization domain [3]. To remedy this, we also present a visual programming environment where the user builds image-space visualization operations using a drag-and-drop interface. The system then generates and compiles matching GLSL [32] shader code that implements the specified operation. We show how this approach can be used to build both existing common information visualization operations, as well as to define new ones.

2 BACKGROUND

In order to set the scene for our general approach to introducing programmable shaders in information visualization, this section will first describe the theoretical foundations of the field. We then discuss the rise of the GPU, its applications to general computing, and its use in scientific and information visualization.

2.1 Foundations of Information Visualization

Shneiderman [35] presented one of the original taxonomies of information visualization. Card and Mackinlay [6] followed up with a study of the morphology of the field’s design space based on Bertin’s semiotics of graphics [2]. These approaches were later unified [7] in the

• Bryan McDonnel is with Purdue University in West Lafayette, IN, USA, E-mail: bmcdonne@purdue.edu.

• Niklas Elmqvist is with Purdue University in West Lafayette, IN, USA, E-mail: elm@purdue.edu.

Manuscript received 31 March 2009; accepted 27 July 2009; posted online 11 October 2009; mailed on 5 October 2009.

For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

concept of the *information visualization pipeline*. Chi [8] presented a refinement of the pipeline consisting of a state model supporting user interaction and data operators. These models will serve as starting points for our own refinement that will support offloading to the GPU.

2.2 The Rise of the GPU

Early computer graphics systems were dominated by fixed-function non-programmable graphics architectures [36]. These implemented a fixed and highly-optimized rendering path, from input 2D and 3D vertices representing graphics primitives, to actual colored pixels on the screen. However, requirements for more flexible shading specified on a per-surface level led to the introduction of shade trees [9] and to early shader definition languages (such as RenderMan) [22].

Fueled mainly by the game development and entertainment industry, graphics hardware then improved rapidly in both performance and programmability to the point where a modern card, such as an NVidia GeForce, has more transistors than an Intel Pentium CPU [36]. In fact, commodity computer graphics chips, more generally known as Graphics Processing Units (GPUs), may be today's most cost-effective computational hardware—their performance growth rate has lately been 2.5-3.0 times a year, which is faster than Moore's Law for CPUs [28].

Controlling the new programmable pipeline is done through machine instructions running on the graphics card, but a number of high-level languages, based on the work by Hanrahan and Lawson [22] as well as Proudfoot et al. [29], have been developed; examples include NVidia's Cg, Microsoft's High-Level Shading Language (HLSL), and the OpenGL shading language (GLSL) [32].

2.3 General-Purpose GPU Computing

Programmable GPUs were obviously designed for graphics processing, but a recent trend has been to press the GPU into service for general computation and algorithms, a method known as *general-purpose GPU computing* (GPGPU) [36]. In this approach, the GPU is regarded as a stream processor, an idea dating as far back as to Fournier and Fussell [19]. Because of the high performance and design of the GPU, this allows for highly parallel and efficient computation. However, the underlying shader language still deals in graphics primitives, which requires shader language expertise and also constitutes a mismatch between the problem and implementation domain [3].

To remedy these problems, a number of GPGPU libraries, such as NVidia's CUDA, ATI's Stream Computing SDK, and the Brook library [3], have been developed to lower the expertise requirements and to provide a better match to the computing domain. Beyond these, there exists a wealth of algorithms and data structures that have been ported to run on the GPU; see Owens et al. [28] for a survey.

2.4 GPUs for Scientific Visualization

The increased programmability of graphics hardware did not go unnoticed in the scientific visualization community—Weiskopf [39] gives a summary of current approaches to using shaders to increase performance and the visual quality of scientific visualizations, and Engel et al. [13] surveys real-time volume rendering techniques using GPUs.

One reason for this quick adoption may be that the data visualized by scientific visualization generally have a spatial 2D or 3D mapping and can thus be easily expressed in terms of the graphics primitives of current shader languages. In general, the current GPU programming model is well-suited to managing large amounts of such spatial data.

However, just as for GPGPU computing, visualization researchers acknowledge and address the two-pronged problem of expertise requirements and conceptual mismatch of writing visualization shaders. AVS [37], dating back to well before programmable graphics hardware

were becoming widely available, allows for building scientific visualizations using small interconnected modules. More recent approaches include block shaders [1], realtime shader generation from code snippets [18], G^2 -buffers for image-space rendering [10], abstract shade trees [27], and dynamic shader generation for multi-volume rendering [31]. Particularly relevant to the framework presented in this paper is Scout [26], a pioneering domain-specific language for dynamically creating visualization shaders. Our ambition in this project is to provide an information visualization equivalent to Scout, albeit with a visual and not a textual programming interface.

2.5 GPUs for Information Visualization

Given the widespread adoption of GPU programmability in the field of scientific visualization, it is surprising that the sister field of information visualization has made so little inroads towards similar adoption. Upon surveying the literature, there are only a few papers that utilize the programmable pipeline of modern graphics hardware, and a general approach for information visualization has yet to be presented.

Existing information visualization systems that do utilize programmable shaders tend to do so as a detail of the implementation, and not a contribution in itself. For example, Johansson et al. [23] uses floating-point OpenGL textures to store data and renders their parallel coordinate display using a GPU pixel shader. Florek and Novotný [16, 17] utilize graphics hardware to improve the rendering performance of scatterplots and parallel coordinates, but do not perform any computations. The ZAME [11] graph visualization system draws multi-value data glyphs using a fixed GPU-shader architecture.

In summary, we are aware of no general method for adopting GPU shaders for information visualization research. Our ambition with this paper is to fill this void by suggesting such a framework, as well as to present a visual programming approach for constructing visualization shaders without requiring special expertise in shader programming.

3 IMAGE-SPACE VISUALIZATION OPERATIONS (IVOs)

The traditional information visualization pipeline consists of three states and three basic transitions between them (Figure 1): starting from a dataset, load the data into structured tables, then mapping the data into visual structures, and finally transforming structures into views that can be displayed on the screen for interpretation by the user. Each of these transformations into the individual states can be user-controlled, closing the feedback loop in the interactive system.

Although traditionally not exposed in an application, virtually all implementations of the pipeline also sport an ultimate image-space transformation where views of visual structures are transformed into color pixels visible on the screen. We propose to refine the pipeline model by exposing an intermediate state where data has been rasterized into the image space of the screen but has not yet been mapped to colored pixels (see the added "Data Image" step in Figure 1). We argue that this last transformation, from data sampled in screen space to actual graphical pixels, form a class of *image-space visualization operations* (IVOs) that represent a previously unrecognized part of the design space of visualization and that are particularly amenable to implementation using programmable graphics hardware.

In this section, we describe a formal model for image-space visualization operations and discuss a basic set of different operation types that this model supports. In the next section, we shall see how these operations can be implemented using current GPU shader languages.

3.1 Model

We define an image-space visualization operation *ivo* as a function that transforms a data tuple $d = (d_1, d_2, \dots, d_n)$ sampled in screen space at position (s_x, s_y) into an RGBA color pixel $p = (p_r, p_g, p_b, p_a)$ for the

corresponding position. Operations are applied to a data stream D of ordered data tuples $d \in D$, where each tuple is independent of others.

The data sampling discussed above is conducted by the actual visualization technique that determines how data-carrying entities should be laid out on the screen. These data entities will then be transformed into actual pixels through a standard *rasterization* algorithm that samples the geometric appearance of the visualization into the pixel grid, akin to how any visualization renders geometric shapes as colored pixels.

Image-space visualization operations (IVOs) are limited to local computations, but they may have access to a global state G that is common to all operations for the specific data buffer, as well as local meta-data M stored as part of the data tuple. The global state G includes colors, filters, and data mappings set by the user or the visualization. Meta-data M includes the screen position (s_x, s_y) , local homogeneous coordinates for the pixel (l_x, l_y) (where $l_* \in [0, 1]$ specifies a local position inside the current graphical object, such as a glyph), and a scale factor s relating the local homogeneous coordinate system to the screen.

Given these definitions, we summarize the function *ivo* as follows:

$$ivo :: I = \langle d, M, G \rangle \rightarrow p = \langle p_r, p_g, p_b, p_a \rangle$$

3.2 Composing Image-Space Operations

The strength of the IVO framework is that an image-space visualization operation *ivo* can be *composed* from several *image-space visualization components* (IVCs) as long as the resulting function composition obeys the above interface, i.e. accepts a data tuple and produces a color pixel. In particular, this entails that the final step in all IVOs is a color mapping from data value to color. However, it does not impose any additional constraints on the components involved.

This composition feature allows for building complex IVOs using a set of pre-defined IVCs. In the following section, we give a basic toolbox of components for building visualization operations in this way.

3.3 Image-Space Visualization Components

While image-space operations are limited in their functionality and scope, they nevertheless represent an important and ubiquitous subset of general visualization. Below is a list of categories for IVCs:

- **Color mapping:** Maps the data tuple d to a color p using some mapping, such as a color scale. Color mappings are typically one-dimensional, i.e. one value of the tuple would be used as input to generate a corresponding color (such as a grayscale or heatmap), but two- (for example, utilizing hue and saturation for an HSB color map) or even three-dimensional color scales are possible, if perhaps not particularly useful.
- **Glyph rendering:** Draws a glyph, such as a miniature barchart, histogram, or line graph, of the contents of the tuple d . The local coordinates l_x and l_y for the pixel in the glyph are used to deduce which color should be assigned to the pixel (e.g., white if the pixel is outside of a particular bar in a barchart, or the bar color otherwise). By switching between different glyph IVOs, the visual representation could be changed instantly. Examples of suitable glyphs are presented in the ZAME [11] paper.
- **Representation switching:** Selects among different visual representations (such as different glyphs of varying complexity) depending on a control metric, such as the amount of visual space devoted to the graphical object. This kind of component could be used to implement semantic zooming in a visualization, switching between different visual representations depending on the current zoom level for the visualization.
- **Filtering:** Filters data by discarding all tuples d that fall outside some range in the data (i.e., producing an empty element \emptyset , or,

alternatively, a fully transparent pixel $p_{trans} = (-,-,-,0)$). Composing several filter IVCs allow for combining filters to produce conjunctions or disjunctions, such as for dynamic queries [40].

- **Computation:** Computes some metrics on the data tuple d , like statistical (averages, medians, or standard deviation), arithmetic, trigonometric, or logarithmic operations on the data.

An example IVO consisting of components in the list above is shown in Figure 2. Using the input data tuple d as a data series, it scales the input (computation), filters out entities outside a specific range (filter), and then transforms the data to a barchart glyph by using the local coordinates (l_x, l_y) (glyph). The output of the barchart component will either be a value if the tuple is part of a bar in the chart, or \emptyset otherwise. This value is mapped to a grayscale pixel (color mapping).

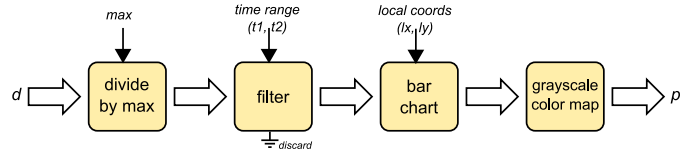


Fig. 2. Simple IVO that scales the value in the input tuple d , filters the data (discarding it if it falls outside a time range (t_1, t_2)), transforms the data point to a barchart glyph of the data series in d , and finally maps the value to grayscale (producing an RGBA pixel p).

3.4 Data Image Feedback

IVOs are designed to only use local scope, so that the mapping of one tuple d is independent of other tuples in the stream D . However, under certain constraints, we may read back the image-space contents of the data image (Figure 1) for analysis and for feedback into the pipeline.

In this way, we can dynamically adapt the IVO depending on the outcome of this analysis. One example of this would be to support query-by-example [15], where the user moves a lens on the visual substrate to select the data ranges to show (i.e., the meaning of this would be “show me all data items that falls within the range of objects I have indicated”). After reading back the values, we can use a filter IVO to discard entities outside the range represented by the selected entities.

Another application of dynamic adaption of the IVO would be to use the data distribution of a selection lens as input for a color mapping IVO, essentially optimizing the color scale for a particular region of interest on the visualization. Figure 7 will give an example of this.

4 GPU IMPLEMENTATION

Given the above theoretical framework, we now show how image-space visualization operators (IVOs) have a natural mapping to implementation as GPU shaders. In order to best understand the underlying concepts, we first discuss a general execution model for GPUs. We then present our approach to implementing IVOs as shaders. We also discuss strategies for reading back and analyzing data images.

4.1 The GPU Shader Programming Model

In order to present our framework for leveraging modern programmable graphics hardware in information visualization applications, it is necessary to provide some background on the current programming model for shader languages such as Cg, GLSL, and HLSL.

In essence, GPU programmable shaders are based on the *stream programming model* [3, 19], where a small program known as a *kernel* is applied to an *input stream* and produce an *output stream*. The key to the high performance provided by graphics cards is that they contain multiple stream processors—modern cards have 128 individual processors or more—capable of executing kernel programs in parallel.

Kernels: Kernels (or *shaders*, as they are known in graphics programming) are implemented either as machine code or in a high-level shader language that is later compiled into machine code. The kernel program is invoked by a stream processor for every element in the input stream and used to produce a new element in the output stream. Because the same kernel is executed in parallel across all stream processors of the graphics card, potentially on a whole stream at once, one instance of a computation can never depend on the result of another. This enforced “embarrassing parallelism” is the key feature that enables the high performance of the graphics card.

Graphics hardware typically supports two different types of shaders (kernels): *fragment* (or pixel) and *vertex shaders*¹. The two types differ in what kind of stream elements they accept and produce. Vertex shaders are invoked once for every 3D vertex passed to the graphics API, whereas fragment² shaders are invoked for every pixel (fragment) to be drawn. Thus, vertex shaders produce vertices (3D points), whereas fragment shaders produce fragments (RGBA values).

Streams and Data Storage: Given that kernels are executed by stream processors on the graphics card, the actual streams are implemented by two-dimensional data arrays called *textures*. Harking back to its graphical origins, where textures were essentially images pasted on top of 3D graphics primitives to add detail (such as a photograph of a brick wall to simulate a brick-like surface), textures were originally limited to RGBA images (i.e. 8-, 16-, 24-, or 32-bit), but modern graphics cards now support floating-point textures. This, in particular, is what has enabled the rise of general-purpose GPU computing [36].

In other words, modern graphics cards also support floating-point arithmetic and storage, although older or less advanced hardware may be limited to half-precision (16-bit) floating point values. In addition to these, shading languages often give support for data types such as integers, vectors, colors, and even arrays of values.

Program Execution: Actually invoking a kernel program on an input stream uploaded to the GPU involves three main steps: (1) selecting the rendering target (the output stream)—either an off-screen texture, for pure computation, or the actual visible framebuffer, for rendering; (2) load the kernel program into the stream processors on the graphics card; and finally (3) render a graphics primitive—typically a quadrilateral of the same size as the input stream—with the input stream as a texture pasted on the surface of the primitive.

For example, to compute the sine of a large number of values, the programmer would fill the input values in a 2D buffer, set the rendering target to another buffer of the same size, install a kernel program that simply computes the sine (using a builtin function in the GPU), and then draw a quadrilateral of the same size while using the input as a texture. Instead of drawing anything on the screen, the shader program will have filled the destination buffer with the result, and the programmer would simply read back the buffer contents into system memory.

Naturally, GPGPU libraries such as Brook [3] abstract away from many of the details of writing shader programs, and instead present a coherent stream programming model based on the concepts of kernels and streams. Our ambition with this work is to provide a similar abstraction, but for the information visualization domain.

4.2 Shaders as IVOs

The image-space visualization operations introduced in Section 3 can be easily implemented as fragment shaders in a current GPU shader language. Figure 3 shows how we can model an input stream D by simply drawing data to the color framebuffer. The transformation from data to pixel will be performed using a fragment shader that implements the corresponding image-space visualization operation.

¹New graphics cards also support a new type called a *geometry shader*.
²Fragments are pixels with additional associated data such as color, depth, texture coordinates, etc, and will be drawn if not discarded during rasterization.

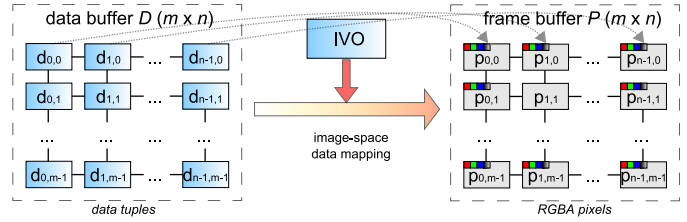


Fig. 3. Image-space mapping using an IVO (image-space visualization operator) shader. Each data tuple $d_{x,y} \in D$ is mapped to its corresponding RGBA pixel $p_{x,y} \in P$ for screen position (x,y) . D is a data buffer rasterized at the same resolution ($n \times m$) as the RGBA framebuffer P . Note that for overlapping visualizations, like 2D scatterplots, there may actually be several data tuples defined for each screen position.)

Fragment shaders are installed globally in the graphics API for a set of primitives. Because visual structures typically have different classes of graphical entities—for example, a node-link diagram consists of edge entities and node entities—we introduce the concept of *layers*, which are a set of graphical entities and a corresponding IVO for transforming them to color pixels. Some layers in a visual structure may serve a purely aesthetic purpose or may not carry image-space data, in which case the IVO will be the identity function.

Figure 4 shows our implementation model for mapping IVOs to the GLSL [32] shader language. Like all fragment shaders, and in keeping with the IVO model, the output is always a colored RGBA pixel. In our prototype implementation, the input data tuple d is passed simply as the color of the primitive. In other words, when drawing the graphical entities for a layer, the program will simply pass image-space data by setting the data as the color for each primitive (such as the marks in a scatterplot). A more scalable solution that would support a large number of data points d_i in d would be to transfer the data as floating-point textures using the multitexture functionality of modern graphics cards. Essentially, instead of reading from a single input color, the shader would sample each texture buffer to retrieve the n data values.

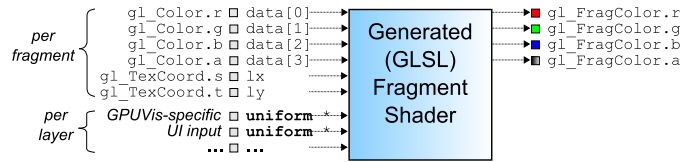


Fig. 4. The GPUVis IVO shader model. Generated GLSL fragment shaders accept data values on a per-fragment level encoded in the OpenGL color and texture coordinate variables, as well as meta-values (e.g. filters, colors, scaling) as uniform arguments on a per-layer level.

Beyond the data inputs in the color argument, the shader IVO will receive the local coordinates of a fragment in a primitive using its texture coordinates. To draw a glyph, the application would simply draw a 2D rectangle (or another graphics primitive) and initialize the texture coordinates from (0,0) to (1,1) for the respective corner points. These values will define the local coordinate system for the primitive.

To illustrate our GLSL shader model, we return to the example IVO in Figure 2. Figure 5 shows GLSL code that implements this particular IVO. Each participating component (IVC) in the IVO has been denoted in the source with a comment and a number (e.g., (1)). Figure 6 shows a screenshot of a 2D scatterplot drawn using this IVO.

4.3 Implementing Data Image Feedback

To implement the data image feedback of Section 3.4, we introduce a G-buffer [33] as an intermediate rendering target implemented using an off-screen framebuffer object (FBO). The OpenGL FBO extension enables us to use the graphics hardware to render the data primitives

```

uniform vec4 scale, minF, maxF;
void main() {
    vec4 d = gl_Color;
    float lx = gl_TexCoord[0].s;
    float ly = gl_TexCoord[0].t;
    // Computation (1)
    d /= scale;
    // Filtering (2)
    if (d[0] < minF[0] || d[0] > maxF[0]
        || d[1] < minF[1] || d[1] > maxF[1]
        || d[2] < minF[2] || d[2] > maxF[2]
        || d[3] < minF[3] || d[3] > maxF[3])
        discard;
    // Barchart glyph (3)
    float v = -1.0;
    if (lx < 0.25 && ly < data[0])
        v = data[0];
    else if (lx < 0.5 && ly < data[1])
        v = data[1];
    else if (lx < 0.75 && ly < data[2])
        v = data[2];
    else if (ly < data[3])
        v = data[3];
    // Color mapping (4)
    if (v == -1.0)
        gl_FragColor = vec4(1, 1, 1, 1);
    else
        gl_FragColor = vec4(v, 0, 0, 1);
}

```

Fig. 5. GLSL fragment shader for an example IVO.

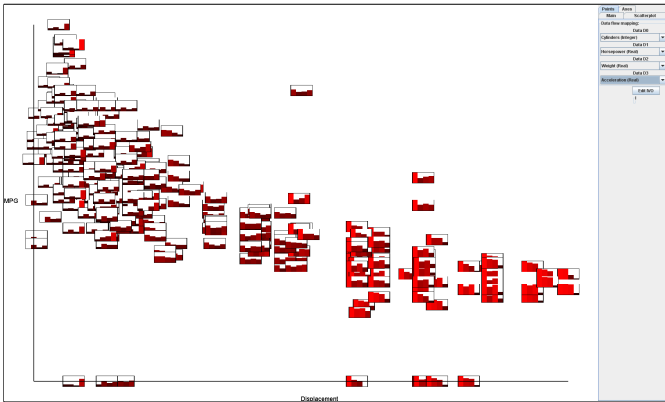


Fig. 6. Scatterplot visualization for a car dataset (9 dimensions, 406 cases) with a barchart glyph IVO.

to a floating-point off-screen render target (one per layer). We can then read back this buffer, analyze it, and use the results to produce the final color framebuffer image. Note that there is no need to redraw the data image until the view changes—instead, we transform the data image into a framebuffer by drawing a single 2D rectangle covering the whole screen and with the data image as a texture.

Figure 7 gives an example of a dynamic colorscale adaption technique implemented using data image feedback and a shader IVO. A similar approach could be used for a query-by-example [15] implementation.

5 DOMAIN-SPECIFIC VISUAL PROGRAMMING ENVIRONMENT

So far in this paper, we have proposed a theoretical extension to the information visualization pipeline and introduced a new class of image-space visualization operations. We have also shown how these operations can be easily mapped to current shader languages, and given examples for the GLSL shader language. However, there are two main obstacles remaining for making this approach generally usable:



Fig. 7. Dynamic colorscale optimization IVO. We draw the data to a G-buffer FBO, read it back, analyze the contents of the sampling lens, and then change the colorscale optimized for the lens contents.

Expertise. Writing shaders requires a great level of programming expertise, increasing the gap between expert developers and visualization professionals [31]; and

Conceptual mismatch. There is a conceptual mismatch between the graphics-centric shader language, and the data-centric visualization operations [3, 26].

In other words, there is a need for a domain-specific language, built on top of the actual shader language, that maps directly to the visualization domain and that does not require shader expertise to use.

5.1 System Architecture

We have developed GPUVis, a visualization environment supporting image-space visualization operations. To fulfill the above need, we provide a visual programming interface for building IVOs.

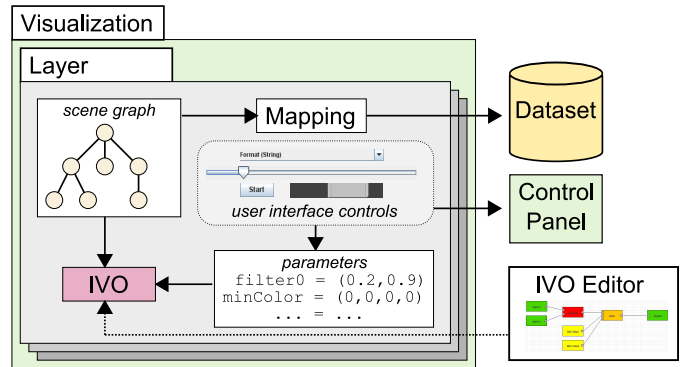


Fig. 8. GPUVis system architecture. Visualizations are loaded as plugins depending on the type of dataset and consists of one or several layers, each with its own IVO. Depending on the parameters exposed by the IVO, the application control panel will be populated with the corresponding GUI controls to specify the parameters.

Figure 8 shows the modular GPUVis system architecture. The system is based around a central dataset structure. Depending on the type of the data the user loads into the dataset, the environment will instantiate a corresponding visualization plugin (i.e., a scatterplot for multidimensional data, a treemap [34] for hierarchical data, a node-link diagram for graphs, etc). Each visualization manages one or more *layers* (as defined above), and each layer contains the graphical primitives making up the visualization and an IVO—editable by the user in a visual IVO editor (described in detail in Section 5.3).

5.2 User Interaction

Interaction is managed by the GPUVis control panel, a tabbed layout with one tab page for each layer. The user can control the data map-

ping between the dataset and the graphical primitives for each layer using controls in the panel. General interaction techniques, such as zoom, pan, and drill-down, are implemented by each visualization.

IVOs in our visualization environment are built using the visual IVO editor. Beyond the actual operation (implemented as a GLSL shader), an IVO also contains a symbol table of parameters and their types (such as color, value, interval, etc). Every time a particular layer's IVO is edited, the system will iterate over the symbol table and create a matching user interface component for controlling the parameter (for example, a color generates a color chooser dialog, a bounded value generates a slider, and an interval generates a range slider [40]).

5.3 Visual IVO Editor

In order to circumvent the need for expert programming skills when constructing custom shader IVOs, the GPUVis system contains a visual IVO editor. This visual programming environment abstracts IVO development into a dataflow-style pipeline composed of *component blocks*, each representing an image-space visualization component (ISVC, see Section 3.3), implemented using a small segment of GLSL shader code. The blocks are arranged and connected to represent the desired image-space visual operation, and a GLSL shader is generated from the graphically described pipeline. This generated shader can then be directly used in the visualization environment.

5.3.1 Background

Visual programming [5, 30] is the use of graphics to specify a program. One of the many uses of visual programming languages (VPLs) is for graphics and visualization. ConMan [21] is a VPL for building graphical applications by connecting components in an approach similar to UNIX pipes. Building on the same design principles is AVS [37], a visual programming system for scientific visualization where each individual module may have an associated user interface component. In related work, Burnett et al. [4] discuss the use of VPLs for interactively steering and modifying scientific computations through visualization.

More recent visualization systems have also been designed around a visual programming paradigm and serve as inspiration for our work. GeoVISTA Studio [25] is a flow-based visual programming environment built using JavaBeans components for geographic visualization. The DataMeadow [12] visual exploration system combines a dataflow VPL with multidimensional visualization techniques and annotation functionality. Finally, Improve [38] is a highly configurable and adaptive visualization system for multiple coordinated views.

5.3.2 Visual Pipeline Construction

Custom IVO creation begins with the visual depiction of the desired operation. As stated above, this is accomplished through the use of objects called component (ISVC) blocks. These blocks represent a few lines of GLSL shader code that perform a distinct function. Each block has inputs and outputs, allowing it to accept data from preceding blocks, manipulate that data according to its particular function, and make the result available for succeeding blocks. Figure 9 shows the anatomy of two different types of blocks.

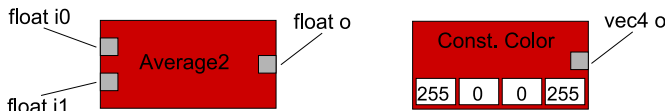


Fig. 9. Sample component blocks. The block on the left computes the average of the 2 input values, while the block on the right allows the pipeline designer to provide the IVO with a constant value.

The specific properties of a given ISVC block (such as number of inputs/outputs and the corresponding GLSL shader code) are organized

in individual XML files. These files are loaded when the editor starts, and are made available to the designer in the pipeline toolbox. The desired blocks can then be dragged from the toolbox and dropped in the pipeline workbench. Once there, the blocks can be arranged and connected to form the envisioned pipeline. Figure 10 shows the visual programming environment as well as a sample pipeline.

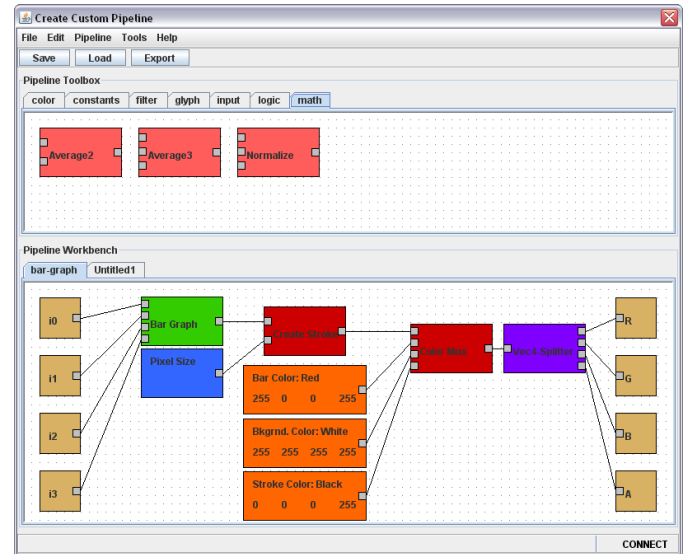


Fig. 10. Visual programming environment and sample shader pipeline. This pipeline corresponds to a shader very similar to the one described in Figure 2, excluding the filtering step. Data flows through the pipeline from left to right. i_0, i_1, i_2, i_3 are the four input data values for the bar graph and R, G, B, A are the four color components of the output pixel.

Since the assembled visual pipeline corresponds to actual GLSL shader code, a certain level of awareness of the underlying IVO model is needed. For example, since an IVO always produces a pixel color value, every pipeline must end with the four elements of that color value (as in Figure 10). Also, the values that each block accepts as inputs, and the values that it makes available as outputs, have data types (float, vec3, vec4, etc). Connected outputs and inputs must have matching data types for the IVO to be valid.

5.3.3 Code Generation

Once the designer is satisfied with the visual representation of the IVO pipeline, actual GLSL code must be generated from the diagram. This is done in two separate stages using a breadth-first traversal:

1. **Dependency Resolution.** If a given block depends on another block's output, the output variable of the first block must be assigned correctly to the input variable of the second block. Dependencies must be propagated through the whole pipeline.
2. **Code Fragment Assembly.** Once all dependencies have been resolved, the individual code fragments must be assembled together in the order specified by the pipeline design.

For a given visualization to have meaningful user interaction, additional information is needed. When the shader code is generated, a symbol table is also built for all uniform variables that the shader depends on. These uniform variables are constant on a per-layer basis, and can be set by the user through the use of variable appropriate components (sliders, range-sliders, color choosers, and spinners).

5.4 Examples

The GPUVis environment currently supports multidimensional scatterplot visualizations as well as basic node-link and treemap [34] vi-

visualizations. We also provide a separate colorscale optimization tool (depicted in Figure 7) built using the principles discussed in this paper.

Scatterplot: As mentioned earlier, Figure 6 shows the scatterplot visualization in the GPUVis system, here depicted with a glyph-based shader for a car dataset. Each glyph is drawn by the CPU and represents a car in the scatterplot. The GPU fills in the contents of the glyph to show additional dimensions beyond the two captured by the orthogonal axes of the plot. By changing the data flow mappings, the data displayed in the barcharts can be changed. Range sliders generated for the IVO allows for filtering out data based on each car’s data.

Node-Link Diagram: Figure 11 shows the GPUVis node-link diagram being used for visualizing a file system hierarchy. The visualization is split into two layers, one for nodes and one for edges. Nodes and their layout are handled by the CPU, but their individual glyphs are drawn by the GPU. In the example, the user has added a glyph IVO to the node layer for drawing a polar barchart on the surface of each node. This glyph can be utilized to display attributes associated with a file, such as file size, time since last modified, and time since creation.

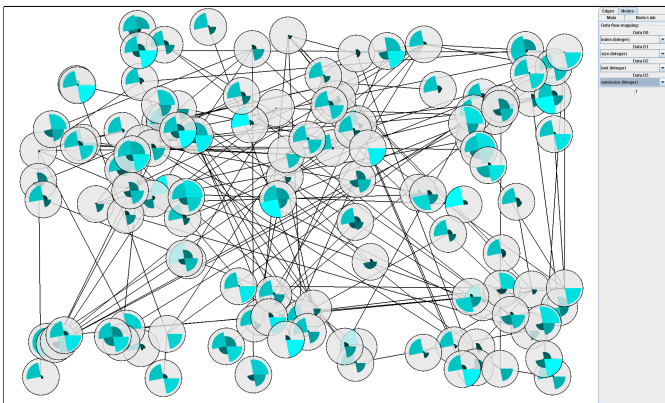


Fig. 11. Node-link visualization with a polar barchart glyph IVO.

Treemap: In Figure 12, the same file system hierarchy is visualized using a standard slice-and-dice treemap, except the application is using a combined barchart and cushion IVO. This enables the file attributes to be displayed on the surface of the barchart, while indicating borders using the cushion fading at the edges of each node area.

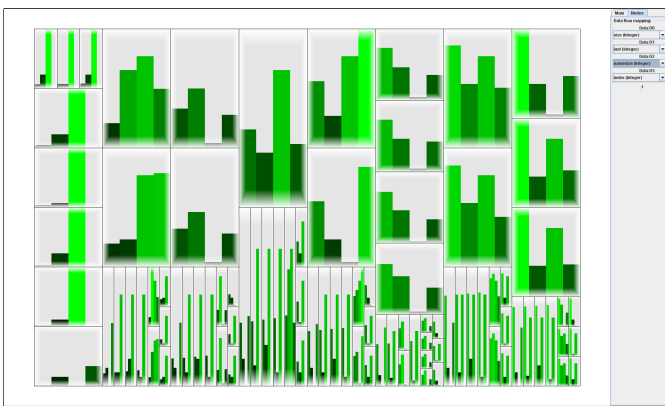


Fig. 12. Treemap visualization using a cushion and barchart IVO.

Performance: Table 1 gives some performance measurements for all three of the above examples where we compare rendering rate (measured as number of frames per second) between a pure OpenGL implementation and a GPU implementation of the same visualization.

As can be seen from these results, the GPU implementation is generally faster, but not significantly so. Only for the relatively complex visual representation in the node-link diagram, where the glyphs are circular, are the benefits of utilizing the GPU obvious.

Visualization	OpenGL	GPUVis
scatterplot (barcharts)	69.5	77.5
node-link diagram (polar barcharts)	19.9	68.0
treemap (barcharts)	89.8	99.8

Table 1. Performance results (frames per second) for the three example visualizations implemented with both pure OpenGL as well as GPUVis (Intel Duo 2.6GHz, 3.5GB RAM, NVIDIA Quadro NVS 140M).

5.5 Implementation Notes

The GPUVis environment was implemented in Java using the JOGL³ OpenGL bindings. It uses the OpenGL shading language (GLSL) [32] for the IVO shader implementations and the OpenGL framebuffer object (FBO) extension for the G-buffer [33] implementation. The visual IVO editor is implemented using Swing and Java2D components.

6 DISCUSSION

In this paper, beyond presenting an approach for how to actually apply programmable shaders to the information visualization domain, we are emphasizing overcoming the expertise and conceptual hurdles of writing shaders, much like Brook [3] for general-purpose computing and Scout [26] for scientific visualization. However, unlike these tools, our prototype implementation uses a visual and not a textual programming interface. The reason for this design choice was to reinforce the information visualization pipeline metaphor that our paper builds on and extends, and this choice also permeates the interaction and visual design of the visual IVO programming editor. However, a textual interface would be a simple extension to the GPUVis system.

There has been some effort in the GPGPU community towards mapping abstract, high-level data structures on the GPU [24]. In particular, recent work on graph layout algorithms [20] utilizes the GPU to achieve high performance. However, this mapping is currently specific to a particular data structure, and no general methodology seems to exist. The approach taken in this paper is different—we do not attempt to map the data structures to the GPU, but rather utilize the highly optimized hardware-accelerated rasterization pipeline for “drawing” the data and perform image-space visualization on the fly. We believe that this is a better approach, at least until GPUs begin supporting more general data structures beyond the current floating-point model.

Many of the techniques and examples presented in this paper may appear somewhat technical and implementation-specific. Nevertheless, it is often implementation details that decide whether an application will scale to massive datasets [14]. Furthermore, this new class of image-space visualization operations identified here fills an important and ubiquitous role in the rendering of any information visualization application, and formalizing this class can only serve to further solidify the foundations of information visualization.

In some fashion, this work can be seen as closing the circle for graphics hardware. Originally developed to draw pixels on the screen, recent developments in computer graphics has turned the GPU into an all-purpose, highly parallel computing device that can be utilized for virtually any purpose. With this work, we propose to turn the emphasis back to rendering, but to retain the computational aspects as well. This could be particularly important for visual analytics, where there is a strong need for computation in the visualization process.

³<http://jogl.dev.java.net/>

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have explored ways to harness programmable GPUs for information visualization. The major hurdle against adopting these methods for information visualization has been the mismatch between abstract data structures such as trees, graphs, and free text and the floating-point model of the graphics hardware. We consider this work to present two major contributions towards such adoption: (1) the theoretical concept of image-space visualization operations (IVOs) that suggest how to utilize shaders for information visualization in the first place, and (2) the practical implementation of a visual programming environment for building shaders that implement the IVO concept.

However, this is only an initial step towards utilizing the full potential of graphics hardware. The approach we take here is limited to image-space operations in the ultimate transformation from data to colored pixels in the visualization pipeline, and does not support offloading whole visualizations or complex datasets to the GPU. In the future, we would like to integrate even more computational components in the visualization pipeline, especially for supporting visual analytics applications. Also, we would like to explore the use of CUDA or OpenCL instead of GLSL as an output language.

ACKNOWLEDGEMENTS

Thanks to members of the AVIZ and INSITU research groups for their feedback on the early stages of this research. This work was conducted under the Purdue SURF 2009 program for undergraduate research.

REFERENCES

- [1] G. D. Abram and T. Whitted. Building block shaders. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 283–288, Aug. 1990.
- [2] J. Bertin. *Semiology of graphics*. University of Wisconsin Press, 1983.
- [3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.
- [4] M. Burnett, R. Hossli, T. Pulliam, B. V. Voorst, and X. Yang. Toward visual programming for steering scientific computations. *IEEE Computational Science & Engineering*, 1(4):44–62, 1994.
- [5] M. Burnett and D. McIntyre. Special issue on visual programming. *IEEE Computer*, 28(3), 1995.
- [6] S. K. Card and J. Mackinlay. The structure of the information visualization design space. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 92–99, 1997.
- [7] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in information visualization: Using vision to think*. Morgan Kaufmann Publishers, San Francisco, 1999.
- [8] E. H. Chi and J. T. Riedl. An operator interaction framework for visualization systems. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 63–78, 1998.
- [9] R. L. Cook. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH 1984)*, pages 223–231, 1984.
- [10] M. Eissele, D. Weiskopf, and T. Ertl. The G²-buffer framework. In *Simulation und Visualisierung 2004 (SimVis 2004)*, pages 287–298, 2004.
- [11] N. Elmqvist, T.-N. Do, H. Goodell, N. Henry, and J.-D. Fekete. ZAME: Interactive large-scale graph visualization. In *Proceedings of the IEEE Pacific Visualization Symposium*, pages 215–222, 2008.
- [12] N. Elmqvist, J. Stasko, and P. Tsigas. DataMeadow: A visual canvas for analysis of large-scale multivariate data. *Information Visualization*, 7:18–33, 2008.
- [13] K. Engel, M. Hadwiger, C. Rezk-Salama, and D. Weiskopf. *Real-time Volume Graphics*. AK Peters, 2006.
- [14] J.-D. Fekete and C. Plaisant. Interactive information visualization of a million items. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 117–124, 2002.
- [15] K. Fishkin and M. C. Stone. Enhanced dynamic queries via movable filters. In *Proceedings of the ACM CHI'95 Conference on Human Factors in Computing Systems*, pages 415–420, 1995.
- [16] M. Florek. Using modern hardware for interactive information visualization of large data. Master's thesis, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, 2006.
- [17] M. Florek and M. Novotný. Interactive information visualization using graphics hardware. In *Poster Proceedings of Spring Conference on Computer Graphics*, 2006.
- [18] N. Folkegard and D. Wesslén. Dynamic code generation for realtime shaders. In *Proceedings of SIGGRAPH*, pages 11–15, 2004.
- [19] A. Fournier and D. Fussell. On the power of the frame buffer. *ACM Transactions on Graphics*, 7(2):103–128, 1988.
- [20] Y. Frishman and A. Tal. Multi-level graph layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, 2007.
- [21] P. E. Haerberli. ConMan: A visual programming language for interactive graphics. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 103–111, Aug. 1988.
- [22] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 289–298, Aug. 1990.
- [23] J. Johansson, P. Ljung, M. Jern, and M. Cooper. Revealing structure in visualizations of dense 2D and 3D parallel coordinates. *Information Visualization*, 5(2):125–136, 2006.
- [24] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Gflit: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, Jan. 2006.
- [25] M. G. M. Takatsuka. GeoVISTA Studio: A codeless visual programming environment for geoscientific data analysis and visualization. *Journal of Computers & Geosciences*, 2002.
- [26] P. S. McCormick, J. T. Inman, J. P. Ahrens, C. D. Hansen, and G. Roth. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *Proceedings of the IEEE Conference on Visualization*, pages 171–178, 2004.
- [27] M. McGuire, G. Stathis, H. Pfister, and S. Krishnamurthi. Abstract shade trees. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, pages 79–86, 2006.
- [28] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.
- [29] K. Proudfoot, W. Mark, S. Tzvetkov, and P. Hanrahan. A real time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001*, pages 159–170, 2001.
- [30] T. I. Robert B. Grafton. Special issue on visual programming. *IEEE Computer*, 18(8), 1985.
- [31] F. Rößler, R. P. Botchen, and T. Ertl. Dynamic shader generation for GPU-based multi-volume ray casting. *IEEE Computer Graphics and Applications*, 28(5):66–77, Sept./Oct. 2008.
- [32] R. J. Rost, J. M. Kessenich, and B. Lichtenbelt. *The OpenGL Shading Language*. Addison-Wesley, 2004.
- [33] T. Saito and T. Takahashi. Comprehensible rendering of 3-D shapes. *Computer Graphics (SIGGRAPH '90)*, 24(4):197–206, 1990.
- [34] B. Shneiderman. Tree visualization with tree-maps: A 2-D space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, Jan. 1992.
- [35] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343, 1996.
- [36] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pages 306–317, 2002.
- [37] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.
- [38] C. Weaver. Building highly-coordinated visualizations in Improve. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 159–166, 2004.
- [39] D. Weiskopf. *GPU-Based Interactive Visualization Techniques*. Springer Verlag, 2006.
- [40] C. Williamson and B. Shneiderman. The dynamic HomeFinder: Evaluating dynamic queries in a real-estate information exploration system. In *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 338–346, 1992.