Eurographics Conference on Visualization (EuroVis) 2017
J. Heer, T. Ropinski and J. van Wijk
(Guest Editors)

*Volume 36 (2017), Number 3*

# Stardust: Accessible and Transparent GPU Support for Information Visualization Rendering

Donghao Ren[1], Bongshin Lee[2], and Tobias Höllerer[1]

[1]University of California, Santa Barbara, United States
[2]Microsoft Research, Redmond, United States

**Abstract**

*Web-based visualization libraries are in wide use, but performance bottlenecks occur when rendering, and especially animating, a large number of graphical marks. While GPU-based rendering can drastically improve performance, that paradigm has a steep learning curve, usually requiring expertise in the computer graphics pipeline and shader programming. In addition, the recent growth of virtual and augmented reality poses a challenge for supporting multiple display environments beyond regular canvases, such as a Head Mounted Display (HMD) and Cave Automatic Virtual Environment (CAVE). In this paper, we introduce a new web-based visualization library called Stardust, which provides a familiar API while leveraging GPU's processing power. Stardust also enables developers to create both 2D and 3D visualizations for diverse display environments using a uniform API. To demonstrate Stardust's expressiveness and portability, we present five example visualizations and a coding playground for four display environments. We also evaluate its performance by comparing it against the standard HTML5 Canvas, D3, and Vega.*

Categories and Subject Descriptors (according to ACM CCS): D.2.2 [Computer Graphics]: Software Engineering—Design Tools and Techniques

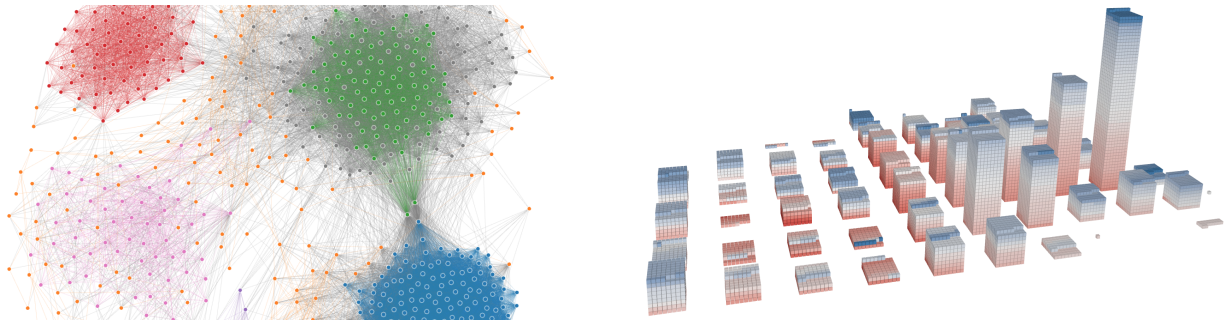
## 1. Introduction

There have been continuous research efforts on visualization libraries and frameworks to facilitate the creation of information visualizations (e.g., [Fek04, HCL05, BH09, BOH11, SWH14, SMWH17]). Thanks to the recent advancement in web technologies (e.g., HTML5, SVG, and D3), and modern browsers that support them, it has become very common to create visualizations for the web environment. Specifically, since its introduction in 2011, D3 has been widely utilized by visualization creators targeting the web. In addition, several higher-level visualization libraries and frameworks (e.g., C3.js [C3j], and NVD3 [NVD]) have been developed using D3 for rendering graphical marks.

However, due to its dependency on the DOM tree and SVG, D3 cannot effectively handle a large number of graphical marks, especially when animating them. For example, in a test on rendering animated scatterplots, D3 can deal with up to about 2,000 points in real-time (faster than 24 frames per second) on a modern personal computer [Rom]. Vega [SWH14] implements its own rendering backends (Canvas or SVG) and performs optimizations on the dataflow model. However, due to the necessity of manipulating or rendering individual marks, it is still hard to achieve high performance on a large number of marks. Visualization libraries based on imperative rendering (e.g., Processing [RF06] and p5.js [RF06]) have a similar performance drawback. While utilizing GPUs can drastically improve rendering performance [ME09, SG15], GPU programming has a steep learning curve and requires a considerable knowledge of the computer graphics pipeline and shader programming. Hence, visualizations that require handling a large number of graphical marks are currently inaccessible to visualization creators who do not have expertise in computer graphics, and thus not yet fully explored by the community.

Immersive analytics has recently begun to gain attention in information visualization [CCC*15, CDK*16, KMLM16]. Compared to desktop or laptop screens, immersive environments allow many more visual elements to be rendered because of the vast virtual spaces they provide. Furthermore, they come in many different form factors such as HMDs, large displays [AEYN11], CAVEs [CNSD*92], and curved/spherical displays. Therefore, not only high performance rendering but also support for diverse display systems are desired in future visualization libraries. Supporting diverse display environments requires a more flexible way of projection and distortion, as well as a uniform representation of both 2D and 3D content.

To address these issues, we designed and developed *Stardust*, a new library that leverages GPU processing power for information visualization rendering while providing a familiar and trans-



**(a)** *Node-link diagram of Facebook users, 747 nodes and 60,050 edges*

**(b)** *Instance-based visualization with 3D stacking, 31,618 items*

**Figure 1:** *Example visualizations created using the Stardust library for GPU-based rendering. For more examples, see Section 5 and visit https://stardustjs.github.io/examples.*

parent programming interface similar to D3. Stardust is designed to support both 2D and 3D visualizations on various display environments through the same programming interface. Stardust consists of a core library and a set of platform libraries. The core library contains a set of predefined visual marks and provides a TypeScript-like programming language for developers to specify custom ones. Mark specifications are compiled to an internal representation that describes input attributes and rendering process. The core library also provides a data-binding API and a set of scales similar to D3's. The platform libraries take the compiled internal representations and the developer-specified data bindings, apply display-environment-specific view transformations, compile them into appropriate GPU shader code (e.g., GLSL, HLSL) and buffers, and finally execute the rendering commands.

The goal of Stardust is not to replace existing visualization libraries and frameworks but to complement them for efficient rendering of graphical marks. Similar to D3.js, we do not intend to provide a new visualization grammar (cf. The Grammar of Graphics [Wil99b] and Vega-Lite [SMWH17]), but to focus on creating a set of building blocks that is easy-to-use, transparent in terms of representation, and portable to multiple platforms.

To evaluate Stardust, we compare it against existing visualization libraries in terms of rendering performance on a modern GPU-equipped PC. Our results show that Stardust achieves 10–100 times speed boost compared to existing libraries when rendering more than 100k graphical marks. We also demonstrate five example visualizations to show its expressiveness (Figure 1, Figure 4, Figure 5) and the support of multiple platforms to show its portability. We develop Stardust as an open-source project (https://stardustjs.github.io) in the hope to further evaluate and improve its usability through adoption and feedback from real users in the future.

## 2. Related Work

### 2.1. Information Visualization Grammars and Frameworks

Many visualization grammars and frameworks have been developed to facilitate the creation of information visualizations. Wilkinson's Grammar of Graphics [Wil99b], ggplot [Wic09], Vega [SWH14], and Vega-Lite [SMWH17] focus on declarative representations, decoupling visualization specification from implementation details. On the other hand, Prefuse [HCL05] provides a set of fine-grained building blocks to allow the creation of custom visualization designs. Protovis [BH09] defines graphical marks and lets developers specify data bindings to their properties. In contrast to the visualization frameworks that define toolkit-specific models, D3 [BOH11] provides a way to directly manipulate elements in the Document Object Model (DOM) based on data attributes. Besides declarative frameworks and libraries, there are also frameworks based on imperative rendering. Processing [RF06], a graphics programming tool widely used by designers and artists, uses an imperative rendering model where developers execute commands to draw graphics. It provides a clear abstraction of drawing commands and an integrated programming experience, and advanced developers can also use low-level APIs such as HTML5 Canvas and OpenGL to draw graphical marks. However, none of these provide easy access to GPU processing power.

In designing a new library that leverages the GPU, we build on the main strengths of D3, one of the most successful libraries widely adopted by developers. Stardust aims for representational transparency to improve understandability, and provides an API that is similar to D3's. On the other hand, instead of mapping data to the *DOM*, Stardust maps data to GPU-rendered marks. While it is not straightforward to specify a new DOM element using D3, Stardust enables developers to specify their own marks with custom input attributes. We explain the commonalities and differences of Stardust's binding API and D3's in Section 3.

### 2.2. GPU-based Visualization Rendering

GPU-based rendering has been widely used in scientific visualization tools. For instance, GPUs are used for accelerating volume rendering [KW03], particle and glyph rendering [FGKR16]. The Visualization Toolkit [SLM04] uses GPUs extensively for scientific visualizations. However, most of these techniques are limited to data that has existing 2D/3D spatial structures. Customization of visual encoding and data binding is limited to the domain of usage. Our work focuses on information visualization where there can be a wide variety of visual mapping and encoding, and existing spatial representations are not guaranteed to exist.

For information visualization, several research projects recently utilized GPUs for accelerating visualizations. For example, SplatterJs [SG15] provides extensive support for scatterplots

with WebGL rendering techniques and the FluidDiagrams [AW14] system implements bar charts, line charts, parallel coordinates, and cone tree visualizations [RMC91]. SandDance [DF15] uses WebGL to efficiently render instance-based visualizations for large datasets and enable smooth transitions between layouts. imMens [LJH13] uses the GPU for efficient data transformation and rendering, but it does not allow customization of GPU-rendered graphical marks. GPU acceleration has also been successfully applied for edge bundling [HvW09]. These visualizations are custom-made, and thus developers of such systems have to write GPU shaders, convert data to buffers, and execute rendering commands. They also provide fixed sets of visualization templates or modules instead of providing developers flexibility in creating their own visualizations.

While GPU-based techniques can significantly improve performance, they are challenging to implement and thus not accessible to developers without expertise in the computer graphics pipeline and shader programming. While compilers (e.g., Sh [MDTP*04] and Brook [BFH*04]) can make GPU acceleration accessible through common programming languages such as C/C++, they are not designed to support information visualizations. Stardust provides visualization developers a simple and familiar programming interface to leverage GPU processing power. We anticipate that developers with some D3 expertise could easily adopt Stardust.

### 2.3. Information Visualization in Novel Display Environments

Researchers have explored large display walls [AEYN11] and immersive environments such as CAVEs [CNSD*92]. The growth of virtual and augmented reality suggests a stronger need for supporting a wide variety of display environments, and efficient rendering is of particular importance in these environments. Although more research is needed to understand when 3D and 2D visualizations are preferable in these environments, enabling more designers to support them can facilitate the exploration of the design space and thus improve our understanding about it.

There are many libraries, frameworks, and engines that support rendering in Virtual Reality (VR) and Augmented Reality (AR), such as FreeVR [SCS13], OpenSG [BO04], Processing [RF06], TechViz [Tec], Unreal Engine [Unr], and Unity Game Engine [Uni]. However, there is little support for building complex information visualizations. Developers have to manually map data items to graphical marks. In a recent demo, Le et al. introduced b3.js [LJB], which supports bar charts, scatterplots, and surface plots in VR displays such as the Oculus Rift. However, it only supports a fixed set of visualizations, and depends on many other libraries. Stardust, in contrast, focuses on solving the data binding problem. It does not provide a set of predefined visualizations, but exposes an easy-to-use API for specifying and manipulating GPU-rendered marks.

Different display environments have different ways of content rendering. For example, some stereoscopic displays employ tilted view frustums; full-surround spherical displays use Omnistereo [PBEP01]. We designed the Stardust's core API to be platform-independent thus allowing developers to customize the platform-dependent part. In our current version, we provide support for WebGL in normal 2D, 3D, and WebVR. By integrating with its rendering library, Stardust visualizations also can be rendered in the AlloSphere [KMWW*14], a full-surround spherical multi-projector display environment. With these display environments we have supported, we demonstrate a web-based coding playground that allows developers to create visualizations using Stardust and run them without modification in multiple display environments including WebGL, Cardboard, HTC Vive, and the AlloSphere.

## 3. Stardust Design

We see Stardust as a complement to D3 instead of a replacement. Stardust is good at rendering a large number of marks and animate them with parameters, while D3 has better support for fine-grained control and styling on a small number of items. For example, to create a scatterplot with a large number of items, we can use D3 to render its axes and handle interactions such as range selections, and use Stardust to render and animate the points.

### 3.1. Design Rationales

**DR1. Representational Transparency and Familiar API:** Representational transparency makes it easy for developers to reason about outcomes (i.e., the resulting visualization), and thus reduces programming and debugging efforts. For example, D3 removes the intermediate layer between user programs and the underlying API by directly binding data to DOM elements. In Stardust, we strive to create an API with similar representational transparency. In addition, we designed the Stardust API to be as close to D3's as possible, while utilizing the GPU for rendering. We believe that the similar API makes it easier for developers to adopt Stardust by transferring their existing knowledge on D3.

**DR2. Declarative Data-Binding over Imperative Evaluation:** D3's imperative evaluation model makes it easier to debug because there are no hidden layers of data flow that complicate the developers' mental model. In GPU programming, however, data binding and shader uniforms have to be fully prepared before one can render anything. This means one cannot render any graphical marks before all attribute bindings are specified. In addition, there is no web inspector for GPU-rendered graphics to provide a similar debugging experience as with D3. Therefore, we decided to use declarative data-binding instead of imperative evaluation.

**DR3. Being Platform Agnostic:** One important design goal of Stardust is to be able to target multiple display environments. Thus, we designed Stardust to be agnostic with respect to platform details including (1) the graphics API (e.g., OpenGL in Unix-based systems and Direct3D in Microsoft Windows) and (2) the rendering and projection method. For example, there are multiple stereoscopic rendering techniques, including tilted view frustums, Omnistereo [PBEP01]. For 3D contents, there are also different shading techniques and post-processing.

### 3.2. API Design

Graphical elements are called "marks" in the Stardust API. The units of operation in Stardust are arrays of marks (e.g., Figure 2-2).

```
(1) let platform = Stardust.platform("webgl-2d", canvasElement, width, height);

(2) let polyline = Stardust.mark.create(Stardust.mark.polyline(), platform);

(3) let positionScale = Stardust.scale.custom(`
      Vector2(
        (R - r) * cos(value) + d * cos((R / r - 1) * value),
        (R - r) * sin(value) - d * sin((R / r - 1) * value)
      ) * 50 + Vector2(250, 250)
    `);
    positionScale.attr("d", 2.19).attr("R", 5).attr("r", 5 * (18 / 41));

(4) polyline.attr("p", positionScale(d => d * 41))
            .attr("width", 2).attr("color", [ 0, 0, 0, 0.3 ]);

(5) polyline.data(_makeRange(0, Math.PI * 2, 10000));

(6) polyline.render();
```

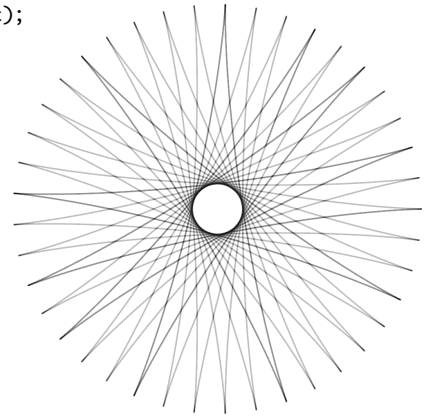


**Figure 2:** *Plotting a parametric function (Hypotrochoid) with Stardust. Left (1) creating a platform object with the given "canvasElement" using the WebGL platform; (2) creating the polyline mark; (3) creating a custom scale for position; (4) setting the attributes for the polyline; (5) assigning data (here we create an array of 10,000 numbers from* 0 *to* 2π*); and (6) rendering the polyline. Right: The resulting plot. Refer to the supplemental material for an animated version of this example.*

Each array of marks has (1) a specification of the mark type; (2) attribute bindings that declare how data items map to marks' input attributes (e.g., Figure 2-4); and (3) an array of data items (e.g., Figure 2-5). Stardust uses a declarative model with lazy evaluation that is different from D3's selection-driven approach where operations are imperatively evaluated (DR2). In Stardust, actual data bindings happen at the `render` call (e.g., Figure 2-6), where Stardust creates GPU shaders and uploads data to GPU buffers for rendering. To be independent of platform details (DR3), Stardust does not rely on any underlying scene graph representation such as the DOM model operated by D3, avoiding the complexity of D3's `enter` and `exit` operations. Updates to the array of data items can be done with array operations.

**Mark Specification:** To make it easy to create basic visualizations, Stardust provides a set of predefined marks, including `circle`, `rectangle`, `line`, `polyline`, and `wedge`. For example, in Figure 2-2, we use the `polyline` mark to plot the parametric function. Because the expressiveness of predefined marks is limited, we provide a mark specification language based on TypeScript [Typ], which is a typed superset of JavaScript (DR1). Custom marks can be created by writing a function with input attributes, intermediate computation steps, and emit statements. For example, the code below creates a custom mark — a range bar with a circle:


```
// Import predefined marks
import { Circle, Line } from P2D;
// Define a utility mark
mark VLine(x: float, y: float) {
  Line(Vector2(x, y - 3), Vector2(x, y + 3), 1);
}
mark RangeBarWithCircle(
  x: float, y: float,
  xmin: float, xmax: float
) {
  // Emit the VLine mark defined above
  VLine(xmin, y);
  VLine(xmax, y);
  // Emit predefined marks, Line and Circle
  Line(Vector2(xmin, y), Vector2(xmax, y), 1);
  Circle(Vector2(x, y), 2);
}
```

The mark specification language exposes the details of marks (DR1). For the marks composed of other marks, developers can follow the function calls until they reach the specification of predefined marks, which are also written in the same language.

Stardust currently supports variables, basic expressions, if-else statements, fixed range for-loops, and function calls. It enables developers to represent a wide range of useful marks including common shapes, parametrized glyphs, Bézier curves, ribbons, wedges, and polylines. More language constructs such as arrays, lambda functions, and classes are left for future implementation work.

**Attribute Binding:** Similar to D3's data-driven selections, Stardust allows mapping data items to marks with user-defined attribute bindings. Attribute bindings can be specified in the same way as D3's `attr` operator (DR1); all D3 scales that return numerical values (e.g., `linear`, `log`, and `time`) as well as layout modules including `force` and `treemap` in D3 can be used in Stardust.

In addition to using D3's scales, Stardust introduces *compilable* scales (DR2). Stardust provides predefined scales including linear, logarithmic, and color interpolation, which have parameters such as `domain` and `range` similar to their counterpart in D3. Besides predefined scales, Stardust allows custom scale expressions and attributes. For example, in Figure 2-3, we declare the positions of the polyline points using the parametric function.

In contrast to D3's immediate scale evaluation at the time of data binding, Stardust's compilable scales are transformed to shader code to be executed in GPU. This allows for adjusting parameters (such as the domain and range of a scale) without having to go through all the data items again and upload updated data to the GPU. Since there is currently no support for introspection in JavaScript, Stardust's scale API is different from D3's. For exam-

ple, in Figure 2-4, the lambda function is inside the Stardust scales, which is different from D3's pattern.

Stardust introduces `Array`-typed attributes, which are implemented as textures in the GPU. Developers can use these objects to store additional data attributes or parameters and reference them in the code. This is very useful when the amount of changing data is small but the number of graphical marks to render is large. For example, in a node-link visualization of a relatively dense graph (e.g., a social network), there are $O(n)$ layout parameters, but usually $O(n^2)$ edges to render. Storing the layout parameters using `Array` objects and referencing them while rendering the edges results in a much better animation performance.

**Rendering, Animation, and Interaction:** After specifying all required attributes and data items, simply calling the mark's `render()` function draws the items. The `platform` object (e.g., Figure 2-1) manages the platform-dependent details such as camera position and projection matrices (DR3).

Attributes that are defined as constant values (including `Array` attributes) can be changed after the `render` call without having to rebuild the GPU shader nor uploading data to the GPU. This helps creating parametric animations such as transition and morphing. For example, the parametric plot in Figure 2 can be animated by changing the `positionScale`'s `d`, `R`, and `r` attributes.

D3 uses events directly from DOM elements, such as `mousedown` and `mouseup`, to support interactions. These events occur on the item level where programmers can easily figure out corresponding marks and data items. However, they are not available in GPU-based graphics APIs including WebGL. Stardust supports item-picking by introducing another pass that renders marks into an item-picking render buffer. We assign a unique index to each mark based on the rendering order, and in a buffer, we use the four 8-bit color channels to encode the index of the rendered mark at each pixel; we can encode up to $2^{32}$ marks. Then, we retrieve the corresponding data item by decoding the pixel's color to convert it back to the index.

## 4. Architecture and Implementation

We have designed Stardust using a modular structure and a platform-independent internal representation (DR3) that acts as an intermediate layer between the developer-facing API and display-environment-specific internal code.

**Modular Structure:** Stardust consists of the `stardust-core` module and a set of platform modules. The core module defines the developer-facing API, built-in marks, and scales, and contains a compiler that converts the mark specification language to Stardust's internal representation. The platform libraries transform internal representations into shader programs, and manage shader uniforms and GPU buffers for rendering. The modular structure allows developers to choose only the parts they need (i.e., core plus desired platforms), and thus minimize the library's download size. Developers could also write support modules for additional mark types, scales, and helper functions. As a demonstration, we have created the `stardust-isotype` module, which supports importing SVG files as isotype [HKF15] marks.

**Compiling to an Internal Representation:** The compiler in the `stardust-core` module transforms mark specifications (Figure 3-1) to Stardust's internal representations (Figure 3-3), which are encoded as JSON objects, and thus can be easily serialized and deserialized (DR3). A mark's internal representation consists of (1) a set of input attributes, which can be bound to data using Stardust's data binding API; (2) a set of internal variables, which store computation states; (3) a set of output attributes; and (4) a list of statements. The statements can be assign statements, conditional statements, loop statements, or emit statements. Conceptually, a mark is rendered by setting the input attributes according to user-specified data bindings, executing the statements, and finally capturing the emitted vertices. Each triplet of consecutive emitted vertices forms a triangle, which is rendered to the screen.

The compiler parses the input code and transforms the resulting abstract syntax tree (AST) to the internal representation by inlining function calls and (optionally) unrolling loops. The compiler also performs a necessary type inference and inserts code for type conversions. When a custom scale is used for a mark attribute, the compiler inserts the scale code and scale attributes to the internal representation, and then assigns the result of the scale to a variable for the mark attribute. In this way, the compiler produces a single piece of internal representation that can be passed to the code and buffer generation step as described below.

**Code and Buffer Generation:** The internal representation serves as a platform-independent representation of marks, from which the platform modules (e.g., `stardust-webgl`) perform the final code generation step to produce GPU shader programs (Figure 3-4,5). Stardust first determines the forms of input attributes. It converts the mark attributes that are specified as immediate values (e.g., `.attr("y", 100)`) to GPU shader uniforms (Figure 3-7), and the mark attributes that are data-driven (e.g., `.attr("x", d => d * 10)`) to vertex attributes along with the corresponding GPU buffer (Figure 3-6). Stardust employs lazy update to prevent sending the same uniforms and buffers multiple times. Stardust re-writes the statements in the internal representation using GPU's shader language (GLSL in the case of WebGL), and executes the shaders and associated uniforms and buffers in the GPU (e.g., glDrawArrays) to render the final geometry.

The platform implementation adds its own geometry transformations. For example, in Figure 3-4, the `stardust_mvp_transform` function is inserted and called to produce the `gl_Position` values. Note that the geometry transformations are not necessarily matrix multiplications. For example, to render content in the AlloSphere, we use its own Omnistereo [PBEP01] per-vertex displacement to achieve stereoscopic effects in all directions. The emit statements can be directly mapped to geometry shader's EmitVertex statements on platforms with geometry shader support. In platforms without geometry shaders (e.g., WebGL and OpenGL ES), Stardust can convert the data format and re-write the shader program as described below.

**Handling Emits in the Absence of Geometry Shaders:** Stardust's internal representation naturally maps to the geometry shader in the computer graphics pipeline. However, there are multiple versions of graphics APIs and shading languages, among which only advanced versions support geometry shaders. For example, WebGL

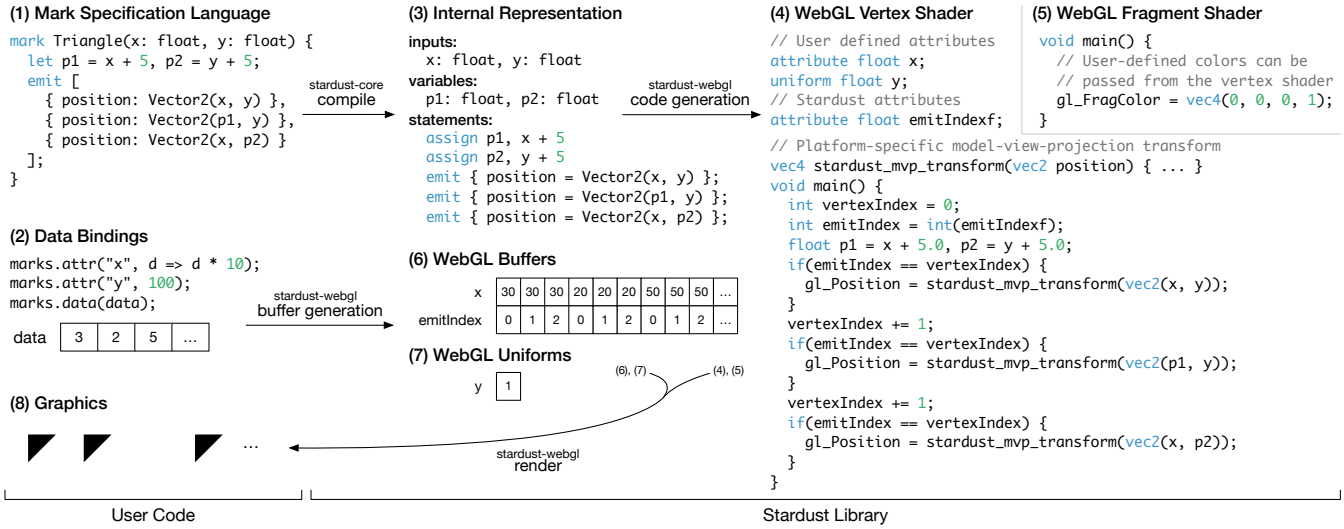



**Figure 3:** *Stardust's rendering process: User code creates mark specifications and data bindings (1, 2). The mark specification is compiled into an internal representation (3). The platform backend, in this case stardust-webgl, converts the internal representation to WebGL shaders (4, 5), and converts the data bindings to WebGL buffers and uniforms (6, 7). Stardust executes the WebGL shaders along with its buffers and uniforms in the GPU to produce the graphics (8).*

has no support for geometry shaders in its current version. Thus, Stardust's platform implementation converts the geometry generation process into a non-generative process when the target platform does not support geometry shaders. Stardust accomplishes this by expanding the mark generation process through program transformation. Originally each mark corresponds to a single data item from which input attributes are derived (e.g., in Figure 3-2, there are three data items, each corresponds to a mark, and the lambda function `d => d * 10` derives a mark's `x` input attribute from the data item). Stardust duplicates the input attributes such that each generated vertex becomes one standalone vertex in the vertex array, increments a vertex emit index on each repetition to differentiate the vertices (e.g., in Figure 3-6, each data item is transformed by the lambda function and then repeated three times, with an added `emitIndex` attribute), and finally re-writes the vertex generation process to use the emit index to determine which vertex to display. Figure 3-4 illustrates how standalone emit statements are transformed, and the code below shows how loops are transformed:

The original for loop:

```
// (input attributes)
for(let i in 2..7) {
    emit F(i);
}
```

Transformed code:

```
// (input attributes)
// emitIndex attribute
// vertexIndex, i variable
vertexIndex = 0;
// For loops without state
// update can be simplified
// to run only one step:
i = emitIndex - vertexIndex + 2;
if(i >= 2 && i <= 7) {
    emit F(i);
}
```

Note that this conversion is performed at the internal representation level so that similar platforms can share it.

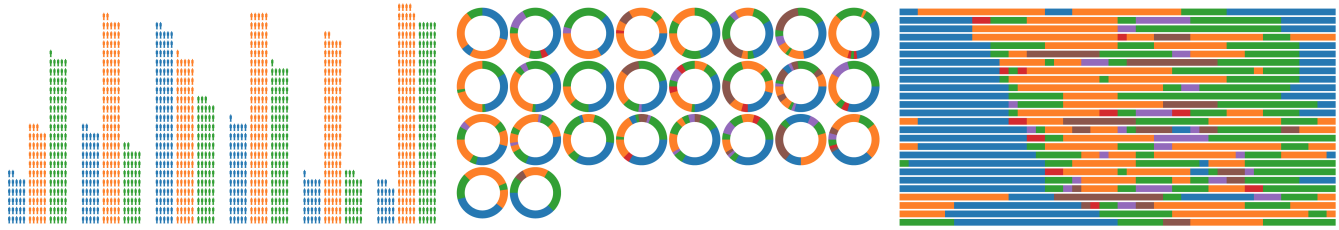
## 5. Examples

With Stardust's predefined marks (circles, lines, areas, and wedges), we can create simple visualizations such as 2D/3D bar charts, scatterplots, line charts, pie charts, area charts, and parallel coordinates. In this section, we show a diverse set of more complex visualization examples to illustrate Stardust's expressiveness, and discuss a coding playground we have created to demonstrate Stardust's portability.

### 5.1. Instance-based Visualization

Researchers have explored instance-based visualizations or unit visualizations, which explicitly represent each data item from the dataset (e.g., [Wil99a, KHDH02, HKF15, DF15]). Here, we illustrate the power of Stardust by animating and interacting with instance-based visualizations.

First, we show a reproduction of animated transitions similar to SandDance [DF15]. The visualization shows voting results in the United States' 2012 election; among 17 attributes we use longitude, latitude, state, and percentage of Obama supporters. In the visualization, we draw the data items as cubes with three different layouts including (1) a scatterplot of longitude and latitude, (2) an instance-based bar chart binned by states, and (3) a 3D stacked bar chart binned by both longitude and latitude (Figure 1b). The cubes are colored by the percentage of Obama supporters. To implement the visualization, we first wrote JavaScript code to compute the bin and within-bin index of each data item for layouts (1) and (3). Then, we wrote marks in the mark specification language. We wrote three functions to compute the positions of cubes based on the three layouts, and in the main function we interpolate between these positions based on the animation time parameter. This example visualization has 31,618 cubes, and the transitions can be rendered at the monitor's refresh rate (60fps) on the computer we used for our performance evaluation (see Section 6).



(a) (b) (c)

**Figure 4:** *Reproduction of Isotype visualization (a) and* "The Daily Routines of Famous Creative People" *in circular layout (b) and linear layout (c) with Stardust.*

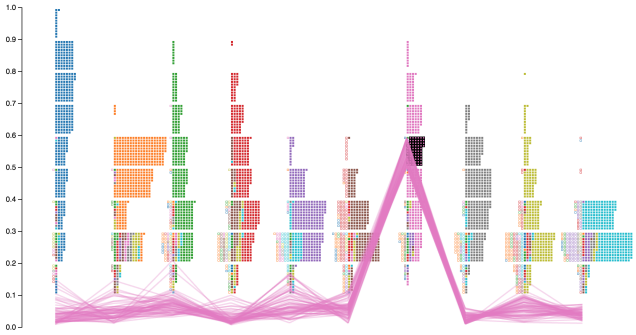


**Figure 5:** *Reproduction of the Squares visualization [RAL*17] using Stardust.*

Instance-based visualizations become more complex when we apply multiple levels of grouping, combine them with other visualizations, and add interaction. We use Stardust to render the Squares [RAL*17] visualization designed for multiclass classifier performance analysis (Figure 5). The Squares visualization design encodes instances as square marks, bins the squares according to their prediction scores, groups them by the predicted classes, and colors them by the labeled classes. Upon selection, parallel coordinates are shown to reveal detailed prediction scores. We use Stardust's item-picking functionality to implement the selection interaction. The version created with Stardust allows parameter changing and interactions at a much more responsive rate than the original D3-based version.

### 5.2. Isotype Visualization

Stardust's custom mark specification allows developers to import marks from other sources. The `stardust-isotype` module can load SVG files and convert them to mark specifications. The converted isotype marks have attributes including `position`, `size`, and `color`, which can be bound to data. Stardust thus supports isotype visualizations [HKF15]. Developers can design isotypes in vector graphics tools and then export them as SVG files. Figure 4a shows our example isotype-based multi-column bar chart. It can perform instance-based animation between two versions (see the supplemental video for the animation).

### 5.3. Morphing between Circular and Linear Timelines

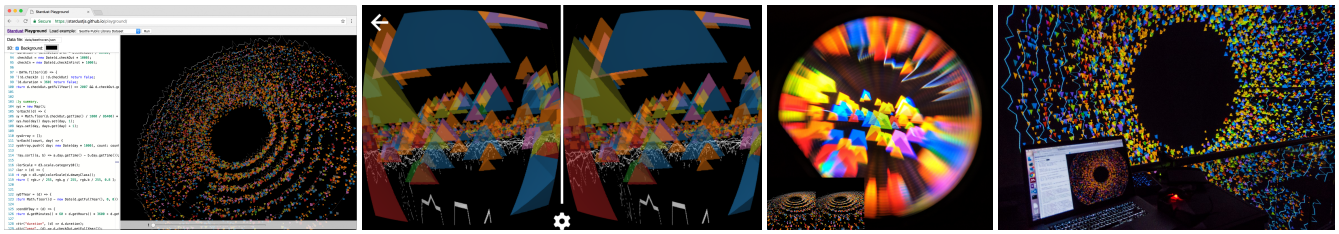
To show Stardust's representational power, we reproduced the visualization of *The Daily Routines of Famous Creative People* [Dai] with Stardust (Figure 4b, Figure 4c), and added a morphing animation between the linear layout and a circular layout as in [BLB*16] (see the supplemental video for the animated transitions). We implemented the marks with Stardust's `Wedge` object. Wedges can also express the linear layout because rectangles are wedges without any curvature. We implemented the morphing animation by linearly interpolating the wedges' parameters between the circular layout and the linear layout. We also used the item index to modulate the animation effect such that animation for different circles/-lines happen at different times. The animated transitions between the two layouts can be easily rendered at 60fps on the computer we used for our performance evaluation, even with synthetic data of 100 times more wedges to render ($100 \times 237 = 23,700$ items).

### 5.4. Real-time Force-directed Graph Visualization

In graph visualizations, algorithms have been developed to accelerate the force-directed layout process [Hu05]. For example, in D3's implementation, a quadtree structure is used to accelerate charge interaction using the Barnes-Hut approximation [BH86]. However, after layout algorithms have been accelerated, rendering becomes the bottleneck for animating graphs. In this example, we use Stardust instead of D3 to render a social-network graph of 747 nodes and 60,050 edges (Figure 1a). Each time the graph layout is updated by the algorithm, we re-assign the data to the nodes and edges, and schedule a re-render of the visualization. The layout algorithm alone can run at 145fps. The Stardust version achieved a significantly better frame rate of 60fps than D3's 1.6fps. Furthermore, this example achieved a frame rate of 21fps on a Google Pixel smartphone with Android 7.1.1 running Google Chrome browser version 56.0.2924.87.

### 5.5. Coding Playground for Multiple Display Environments

We have created a coding playground, which enables developers to write code in a web-browser and run it *unmodified* in four different display environments; 1) the browser itself, 2) mobile VR with Cardboard viewers, 3) HTC Vive, and 4) the AlloSphere [KMWW*14], a full-surround immersive VR environment (Figure 6). This playground is a starting point for building a cross-platform visualization authoring environment. For example, in the future we can explore how to easily design visualizations that can adapt to different levels of processing powers by using aggregation or subsampling techniques: we can show an overview visualization in mobile devices, while showing individual data items and



**(a)** *Browser* **(b)** *Cardboard* **(c)** *HTC Vive* **(d)** *AlloSphere*

**Figure 6:** *A sample visualization created with Stardust in the coding playground. Developers can create 2D/3D visualizations for three different display environments with the same code. (a, b) run in browsers with WebGL; (c, d) run in the AlloSphere's Allofw framework with OpenGL 3.3; and (d) uses Omnistereo for a* 360° *stereo effect.*

allowing multiple coordinated views in immersive environments. Stardust provides a simple visualization representation for various levels of graphics hardware and rendering process behind different display environments.

We implemented the coding playground's user interface using TypeScript and React. It embeds the Monaco text editor for syntax-highlighting and intellisense features. The visualizations in the web browser are rendered in `iframe` elements to minimize the interference between developer code and playground code. In the AlloSphere, we created a web server to receive code and commands from the user interface and coordinate 13 rendering machines that power 26 projectors.

## 6. Performance Evaluation

We evaluate Stardust's performance in terms of the initialization and rendering times on three types of visualizations.

### 6.1. Conditions

We compare Stardust against D3, Vega (Canvas backend), and the standard HTML5 Canvas. D3 is one of the most commonly used libraries in web-based visualizations. Vega is a very promising declarative visualization grammar with a performance-optimized reactive dataflow architecture [SWH14, SMWH17]. We choose Vega's Canvas backend because it is measured to be more efficient than the SVG backend. The HTML5 Canvas is the basis of many visualization libraries such as Processing that are not SVG-based. We implement three visualizations that use common mark types in information visualization: (1) scatterplots with points, glyphs, and isotypes; (2) parallel coordinates with lines and polylines; and (3) node-link graphs with points and lines. The visualizations we created with D3 adhere to common D3 programming patterns, and the ones with Canvas are written to access Canvas' API as directly as possible to eliminate any performance overhead introduced by extra representational layers. For example, to compute an `x` position, we choose `x = a * data + b` over `x = scale(data)` because `scale(data)` introduces an extra function call.

To simulate animation and layout updates, we animate these visualizations by randomly changing scales for the scatterplots and parallel coordinates and shifting node positions for the graphs. We run these visualizations with sizes ranging from 100 to 1M (the size of a graph is measured as the number of edges, which dominates the rendered marks). The resolution of the visualizations is set to $1,000 \times 1,000$. We run our performance measurements with an iMac Retina 5K 27-inch Late 2014 model with the macOS Sierra 10.12.1. The machine has a 4GHz Intel Core i7 processor with 32GB 1600MHz DDR3 memory, and an AMD Radeon R9 M290X graphics card with 2GB memory. The benchmark is performed on the Google Chrome browser version 55.0.2883.75 (64-bit).

### 6.2. Results

It is important to differentiate two types of rendering times: (1) initializing and rendering the first frame, which directly influences the page load time; and (2) rendering subsequent frames in response to parameter updates, which is crucial for parametrized animation. Because modern web browsers commonly use background threads for graphics rendering, especially for WebGL-based rendering, it is not possible to measure frame time directly by timing the API calls. We used the `requestAnimationFrame` API to measure the time duration between adjacent frames and among multiple frames for initialization and rendering, respectively. This approach, however, has a minimum measurable unit, which is the refresh rate of the monitor because browsers wait for the next monitor refresh before performing an "animation frame" when rendering is faster than the refresh rate. On the machine we tested with, this minimum unit is 16.7ms. Therefore, all values we report here are more than 16.7ms whose corresponding frame rate is 60fps. To model parameter updates during rendering, the rendering time is measured by averaging 30 frames with random visual and layout parameters. We also run each trail 10 times and take the average to minimize performance fluctuations. To avoid potential garbage-collection delays, we restart the browser for every trail larger than 10,000 items.

Our performance simulation results show that Stardust is faster in both initialization and rendering for large numbers of marks (Figure 7). The time for initialization and rendering the first frame is around 2x faster than Canvas, and for subsequent frames Stardust is 10–100x faster than Canvas, Vega, and D3. However, Stardust's initialization time is slower than Canvas, D3, and Vega for small numbers of marks because Stardust has an overhead of shader compilation. In addition, we observed that while canvas is faster than D3 for the scatterplot and graph, it is slower than D3 for rendering the parallel coordinates. The fact that D3's render time for parallel coordinates looks quadratic and is slower than its initialization plus render time is notable. We suspect that this is caused by the Google Chrome browser's internal mechanisms for handling updates to the `d` attributes on SVG `path` elements.

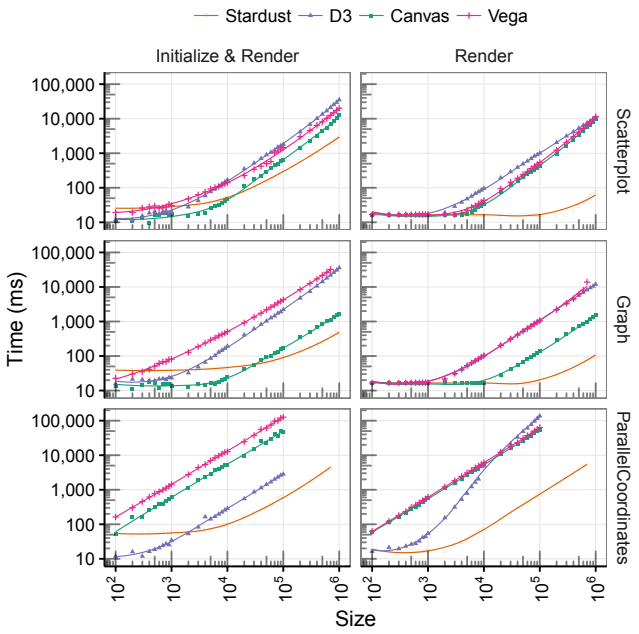



**Figure 7:** *Performance evaluation results plotted in log-log scales.*

## 7. Discussion and Future Work

Stardust is inspired by the observation of media artists creating 3D data visualizations that render and animate a large number of data-driven glyphs. This was very difficult in Processing; drawing the glyphs one by one was too slow even with WebGL/OpenGL, but putting them altogether into a mesh object made it impossible to parametrically animate the glyphs without using a vertex shader. We initially created a library that supports a set of predefined marks and D3-like data bindings. However, creating a new mark required a considerable knowledge of the library and shader programming, and geometry shaders used in the initial version is not currently available in web browsers. Stardust's internal representation and shader code generation architecture contribute to its expressiveness and portability. Together, they support a simple and uniform API while hiding complex platform-specific implementation details (e.g., the geometry generation process).

In this section, we discuss Stardust's limitations which form interesting directions for future work.

**Pixel-space Specifications:** Like D3, the Stardust API currently only specifies geometry with a few styling options including color, opacity, and shading options. Although these cover a large set of information visualizations, other visualizations require advanced styling of marks such as pattern-encoded marks and textured marks. Therefore, it would be useful to extend Stardust to support pixel-space specifications.

**Per-mark State:** To be platform-agnostic, Stardust only abstracts the data-driven geometry generation process without relying on scene-graph models, which naturally keep mark states in nodes. Because Stardust's geometry generation process on GPU is currently stateless, it cannot provide a support for some of the D3 functionalities such as filtered selections and enter/exit selections. Only data-bindings that are specified as immediate values can be changed without having to re-compile shaders and re-upload buffers to GPU. Therefore, it is hard to implement animations that require per-mark state recording such as particle system simulations in Stardust. Having an API support for per-mark state storage, update, and retrieval is important for future work.

**GPU Support for Data Transformation and Aggregation:** Stardust currently supports only GPU-based rendering. Similar to D3, to use Stardust, data has to be aggregated so that each data item *independently* maps to a mark. For example, in our instance-based visualization example (Section 5.1), we have to compute the binning in JavaScript and assign a bin and a within-bin index for each data item. In the future, we would like to investigate how to support GPU-based data aggregation and transformation (e.g., [LJH13]).

**Optimization:** As a first step, we primarily address the problem of simple and easy specification. As we have not yet implemented optimization techniques, Stardust currently relies on the shader compiler's optimizations. For future versions of Stardust, we plan to explore optimization techniques.

**Further Evaluation:** Given Stardust's simplicity and similarity to D3's API, we anticipate it can be easily adopted by visualization creators. As a public deployment could lead to more meaningful evaluations, we constructed a website that provides online code examples and documentation to help developers get started. We hope to get feedback from visualization creators and improve the library.

## 8. Conclusion

We have designed and implemented Stardust, a GPU-based library for information visualization, as an open-source project (https://stardustjs.github.io). It provides a simple programming interface while targeting both regular and novel display environments. Stardust enables the creation of a variety of GPU-powered visualizations as demonstrated by our examples. It achieves a significant performance boost compared to the standard HTML5 Canvas, D3, and Vega on a large number of graphical marks.

## 9. Acknowledgments

We thank Andrés Cabrera and JoAnn Kuchera-Morin for the discussion and support in the AlloSphere. This work was in part supported by ONR grants N00014-14-1-0133 and N00014-16-1-3002.

## References

[AEYN11] ANDREWS C., ENDERT A., YOST B., NORTH C.: Information visualization on large, high-resolution displays: Issues, challenges, and opportunities. *Information Visualization 10*, 4 (2011), 341–355. 1, 3

[AW14] ANDREWS K., WRIGHT B.: FluidDiagrams: Web-based information visualisation using JavaScript and WebGL. In *EuroVis - Short Papers* (2014), Elmqvist N., Hlawitschka M., Kennedy J., (Eds.), The Eurographics Association. 3

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)* (2004), vol. 23, ACM, pp. 777–786. 3

[BH86] BARNES J., HUT P.: A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature 324*, 6096 (1986), 446–449. 7

[BH09] BOSTOCK M., HEER J.: Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1121–1128. 1, 2

[BLB*16] BREHMER M., LEE B., BACH B., HENRY RICHE N., MUNZNER T.: Timelines revisited: A design space and considerations for expressive storytelling. *IEEE Transactions on Visualization and Computer Graphics PP*, 99 (2016), 1–1. 7

[BO04] BURNS D., OSFIELD R.: Tutorial: Open scene graph. In *IEEE Virtual Reality 2004* (2004), IEEE, pp. 265–265. 3

[BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D3: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2301–2309. 1, 2

[C3j] C3.js: D3-based reusable chart library. http://c3js.org/, accessed Dec. 3, 2016. 1

[CCC*15] CHANDLER T., CORDEIL M., CZAUDERNA T., DWYER T., GLOWACKI J., GONCU C., KLAPPERSTUECK M., KLEIN K., MARRIOTT K., SCHREIBER F., ET AL.: Immersive analytics. In *Big Data Visual Analytics (BDVA), 2015* (2015), IEEE, pp. 1–8. 1

[CDK*16] CORDEIL M., DWYER T., KLEIN K., LAHA B., MARRIOT K., THOMAS B. H.: Immersive collaborative analysis of network connectivity: CAVE-style or head-mounted display? *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2016), 441–450. 1

[CNSD*92] CRUZ-NEIRA C., SANDIN D. J., DEFANTI T. A., KENYON R. V., HART J. C.: The CAVE: Audio visual experience automatic virtual environment. *Communications of the ACM 35*, 6 (1992), 64–72. 1, 3

[Dai] The daily routines of famous creative people. https://podio.com/site/creative-routines/, accessed Dec. 3, 2016. 7

[DF15] DRUCKER S., FERNANDEZ R.: *A Unifying Framework for Animated and Interactive Unit Visualizations*. Tech. rep., Microsoft Research, 2015. 3, 6

[Fek04] FEKETE J.-D.: The InfoVis toolkit. In *Proceedings of the IEEE Symposium on Information Visualization* (2004), IEEE, pp. 167–174. 1

[FGKR16] FALK M., GROTTEL S., KRONE M., REINA G.: *Interactive GPU-based Visualization of Large Dynamic Particle Data*, vol. 4 of *Synthesis Lectures on Visualization*. Morgan & Claypool Publishers, San Rafael, CA, 2016. 2

[HCL05] HEER J., CARD S. K., LANDAY J. A.: Prefuse: A toolkit for interactive information visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2005), CHI '05, ACM, pp. 421–430. 1, 2

[HKF15] HAROZ S., KOSARA R., FRANCONERI S. L.: ISOTYPE Visualization: Working memory, performance, and engagement with pictographs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2015), CHI '15, ACM, pp. 1191–1200. 5, 6, 7

[Hu05] HU Y.: Efficient, high-quality force-directed graph drawing. *Mathematica Journal 10*, 1 (2005), 37–71. 7

[HvW09] HOLTEN D., VAN WIJK J. J.: Force-directed edge bundling for graph visualization. In *Proceedings of the 11th Eurographics / IEEE - VGTC Conference on Visualization* (2009), EuroVis '09, The Eurographs Association & John Wiley & Sons, Ltd., pp. 983–998. 3

[KHDH02] KEIM D. A., HAO M. C., DAYAL U., HSU M.: Pixel bar charts: A visualization technique for very large multi-attribute data sets. *Information Visualization 1*, 1 (Mar. 2002), 20–34. 6

[KMLM16] KWON O.-H., MUELDER C., LEE K., MA K.-L.: A study of layout, rendering, and interaction methods for immersive graph visualization. *IEEE Transactions on Visualization and Computer Graphics 22*, 7 (2016), 1802–1815. 1

[KMWW*14] KUCHERA-MORIN J., WRIGHT M., WAKEFIELD G., ROBERTS C., ADDERTON D., SAJADI B., HÖLLERER T., MAJUMDER A.: Immersive full-surround multi-user system design. *Computers & Graphics 40* (2014), 10–21. 3, 7

[KW03] KRUGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *Proceedings of the 14th IEEE Visualization* (2003), VIS '03, IEEE Computer Society, p. 38. 2

[LJB] LE H., JOSHI A., BETKE M.: b3.js: A library for interactive web data visualizations in virtual reality. https://github.com/huyle333/b3, accessed Dec. 3, 2016. 3

[LJH13] LIU Z., JIANG B., HEER J.: imMens: Real-time visual querying of big data. *Computer Graphics Forum 32*, 3 (2013), 421–430. 3, 9

[MDTP*04] MCCOOL M., DU TOIT S., POPA T., CHAN B., MOULE K.: Shader algebra. *ACM Transactions on Graphics 23*, 3 (2004), 787–795. 3

[ME09] MCDONNEL B., ELMQVIST N.: Towards utilizing GPUs in information visualization: A model and implementation of image-space operations. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1105–1112. 1

[NVD] NVD3: Reusable charts for D3.js. http://nvd3.org/, accessed Dec. 3, 2016. 1

[PBEP01] PELEG S., BEN-EZRA M., PRITCH Y.: Omnistereo: Panoramic stereo imaging. *IEEE Transactions on Pattern Analysis and Machine Intelligence 23*, 3 (Mar. 2001), 279–290. 3, 5

[RAL*17] REN D., AMERSHI S., LEE B., SUH J., WILLIAMS J. D.: Squares: Supporting interactive performance analysis for multiclass classifiers. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 61–70. 7

[RF06] REAS C., FRY B.: Processing: Programming for the media arts. *AI & Society 20*, 4 (2006), 526–538. 1, 2, 3

[RMC91] ROBERTSON G. G., MACKINLAY J. D., CARD S. K.: Cone Trees: Animated 3D visualizations of hierarchical information. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (1991), CHI '91, ACM, pp. 189–194. 3

[Rom] ROMERO M.: SVG performance test. http://bl.ocks.org/mjromper/95fef29a83c43cb116c3, accessed Dec. 3, 2016. 1

[SCS13] SHERMAN W. R., COMING D., SU S.: FreeVR: Honoring the past, looking to the future. In *Proc. SPIE, The Engineering Reality of Virtual Reality* (Mar. 2013), vol. 8649. 3

[SG15] SARIKAYA A., GLEICHER M.: Using WebGL as an interactive visualization medium: Our experience developing SplatterJs. In *Proceedings of the Data Systems for Interactive Analysis Workshop* (Oct. 2015), DSIA '15, IEEE. 1, 2

[SLM04] SCHROEDER W. J., LORENSEN B., MARTIN K.: *The visualization toolkit*. Kitware, 2004. 2

[SMWH17] SATYANARAYAN A., MORITZ D., WONGSUPHASAWAT K., HEER J.: Vega-Lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 341–350. 1, 2, 8

[SWH14] SATYANARAYAN A., WONGSUPHASAWAT K., HEER J.: Declarative interaction design for data visualization. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (2014), UIST '14, ACM, pp. 669–678. 1, 2, 8

[Tec] TechViz. http://www.techviz.net/, accessed Dec. 3, 2016. 3

[Typ] TypeScript: JavaScript that scales. https://www.typescriptlang.org/, accessed Dec. 3th, 2016. 4

[Uni] Unity Game Engine. https://unity3d.com/, accessed Dec. 3, 2016. 3

[Unr] Unreal Engine. https://www.unrealengine.com/, accessed Dec. 3, 2016. 3

[Wic09] WICKHAM H.: *ggplot2: elegant graphics for data analysis*. Springer Science & Business Media, 2009. 2

[Wil99a] WILKINSON L.: Dot plots. *The American Statistician 53*, 3 (1999), 276–281. 6

[Wil99b] WILKINSON L.: *The Grammar of Graphics*. Springer-Verlag, 1999. 2