DXR: A Toolkit for Building Immersive Data Visualizations

Ronell Sicat, Jiabao Li, JunYoung Choi, Maxime Cordeil, Won-Ki Jeong, Benjamin Bach, and Hanspeter Pfister

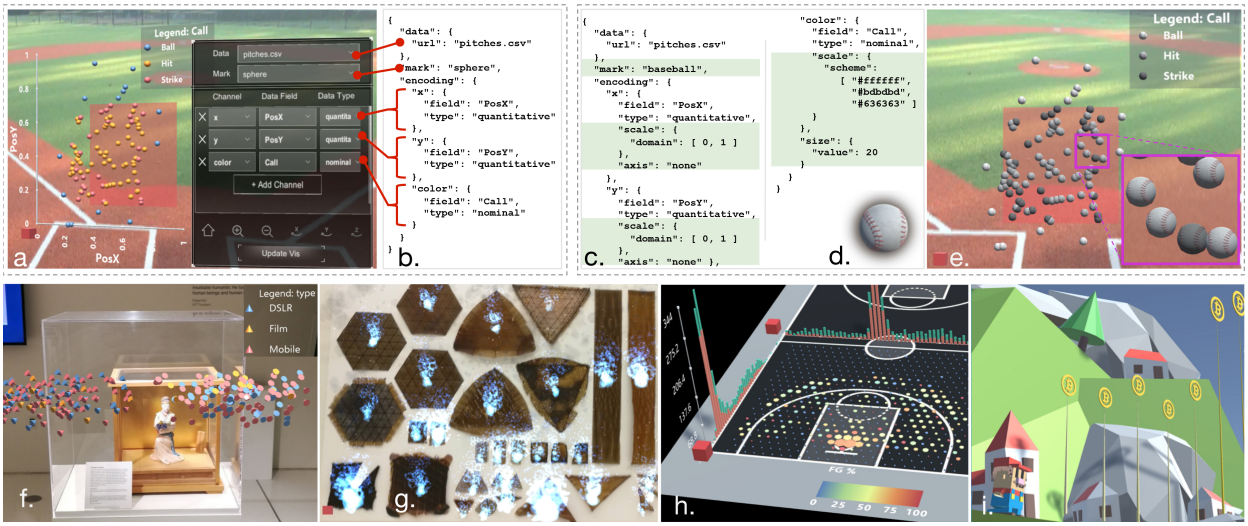

Fig. 1. DXR enables rapid prototyping of immersive data visualizations: (b,c) declarative specifications concisely represent visualizations; (a:right) DXR's graphical user interface (GUI) within the virtual world enables quick iteration over visualization parameters such as data sources, graphical marks, and visual encodings; (b) the GUI modifies the underlying design specifications; (c) specifications can be fine-tuned by the designer in a text editor; (d) the designer can add 3D models as custom graphical marks to achieve (e) novel immersive visualization designs. Example visualizations built using DXR: (f) a 3D vector field plot showing locations of photographs of an exhibit; (g) flames representing the remaining lifetime of real-world organic materials as they decay; (h) bar charts and scatter plots embedding sports data in a virtual basketball court; and (i) coins showing Bitcoin prices in a 3D game.

Abstract—This paper presents DXR, a toolkit for building immersive data visualizations based on the Unity development platform. Over the past years, immersive data visualizations in augmented and virtual reality (AR, VR) have been emerging as a promising medium for data sense-making beyond the desktop. However, creating immersive visualizations remains challenging, and often require complex low-level programming and tedious manual encoding of data attributes to geometric and visual properties. These can hinder the iterative idea-to-prototype process, especially for developers without experience in 3D graphics, AR, and VR programming. With DXR, developers can efficiently specify visualization designs using a concise declarative visualization grammar inspired by Vega-Lite. DXR further provides a GUI for easy and quick edits and previews of visualization designs in-situ, i.e., while immersed in the virtual world. DXR also provides reusable templates and customizable graphical marks, enabling unique and engaging visualizations. We demonstrate the flexibility of DXR through several examples spanning a wide range of applications.

Index Terms—Augmented Reality, Virtual Reality, Immersive Visualization, Immersive Analytics, Visualization Toolkit.

1 Introduction

Immersive technologies such as augmented and virtual reality, often called extended reality (XR), provide novel and alternative forms of representing, interacting, and engaging with data and visualizations [45]. The range of applications that benefit from stereoscopy, augmented reality, natural interaction, and space-filling immersive visualizations is growing, including examples in information visualization [44, 48], scientific visualization [68], immersive storytelling [14, 57, 60], immersive workspaces [50], and embedded data representations [36, 51, 72]. Fueled by the recent increase in affordable AR and VR devices, immersive visualizations have come into focus for many real-world applications and should be meant to being designed and created by a range of people not necessarily trained in XR development.

Building applications and prototyping visualization designs for immersive environments remains challenging for many reasons. It is a craft that naturally requires knowledge of concepts and technology from data visualization and analytics, 3D computer graphics, AR, and VR, as well as human-computer interaction, and human factors. Not only does this hinder fast prototyping and design exploration, especially in a real environment [36], but it creates a high bar for novice developers without background in any of these areas. On the other hand, the success and wide adoption of D3 [43], Vega-Lite [66], and VTK [29] have shown how visualization-specific toolkits and languages empower development, design, and dissemination. We believe it is timely to think about user-friendly tool-support for immersive visualizations.

In this paper, we present *DXR*, a toolkit for rapidly building and prototyping **D**ata visualization applications for e**X**tended **R**eality. DXR is based on the Unity development platform [24]. While Unity enables XR development, it still has limited support for rapid prototyping of

- *R. Sicat and H. Pfister are with the Harvard Visual Computing Group.*
- *J. Li is with the Harvard Graduate School of Design.*
- *B. Bach is with the School of Informatics at Edinburgh University.*
- *M. Cordeil is with the Immersive Analytics Lab at Monash University.*
- *J. Choi, and W.-K. Jeong are with Ulsan National Institute of Science and Technology.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

visualization applications. Currently, designers must write low-level code using C# or JavaScript to parse data, manually instantiate objects, create the visual mappings, bind data to visual object properties, and to implement interaction, placement, and propagation of data updates. Furthermore, iterative design changes require tuning of low-level code, which prohibits quick prototyping. In turn, DXR provides a high-level interface for constructing and adapting pre-configured visualizations (Fig. 1) in Unity. It uses a Vega-Lite inspired grammar to specify a visual mapping for data imported through DXR data loading routines. Changes to the visual mapping automatically update the visualization in XR while the developer wears the AR or VR headset and sees the changes in real-time [54]. Advanced users can edit the Vega-Lite-like design specifications in JavaScript Object Notation (JSON) in any text editor. Eventually, designers can create custom graphical marks and visual channels that leverage the wide variety of Unity prefabs for building unique and engaging designs.

DXR comes with a library of reusable pre-defined visualizations such as scatter plots, bar charts, and flow visualizations, which can be connected through filtering and linking. DXR visualizations are Unity GameObjects that are compatible with the full feature set of the Unity development environment and associated libraries, e.g., for object tracking and placement, interaction, styling, etc. Interactive DXR applications can be exported to a variety of platforms, including AR on Microsoft HoloLens [15], and VR headsets.

DXR aims to allow a wider community to explore and design immersive data visualizations. Use cases for DXR range from individual interactive 2D or 3D visualizations, immersive multi-visualization workspaces, to embedded data representations [36, 72] that apply XR technology inside museums, sports arenas, and scientific labs, to name a few. DXR is open-source, freely available at `sites.google.com/view/dxr-vis`, with a range of well-documented reusable examples.

2 Background and Related Work

2.1 Applications of Immersive Visualization

Immersive visualizations have been built for most common visualization types, including scatter plots [35], parallel coordinates [44], networks [49], and sports analytics applications [30]. ImAxes [48] implements scatter plots and parallel coordinates that are explorable and reconfigurable through embodied interaction.

Beyond exploration, AR and VR are often used as a medium for experiencing data-driven presentations and storytelling. For example, LookVR [14] turns bar charts into virtual walls that can be climbed in VR. Beach [60] virtually puts users in a room with dangerously increasing sea levels to educate them about climate change. An application by the Wall Street Journal [57] lets users virtually walk along a line chart like a staircase to experience the rise and sudden fall of the Nasdaq index during a stock market crash. All these examples present data-driven scenes [38] that allow end-users to relate the data to real-life experiences for better engagement and impact. These visualizations are typically created by artists, storytellers, designers, and domain experts who had to invest time to learn visualization and XR development.

Many more examples lend themselves to an exploration through immersive technology motivated by better spatial perception, a larger display space, or bringing together physical referents and their data [36, 69, 72]. Coupled with immersive displays, immersive interactions beyond the mouse and keyboard allow natural and direct interaction with data in AR [35] or VR [68] environments. Other studies have shown the benefit of immersion for collaborative data analysis [49, 50].

2.2 Authoring Immersive Visualizations

The most common platform to develop XR applications is Unity [24], a game engine with a large community and a range of modular and additional *assets* such as 3D models, and scripting libraries. For AR, additional frameworks exist to support object tracking and rendering in general, e.g., ARToolkit [6], Vuforia [31], or for specific platforms, e.g., ARCore for Android [4], ARKit for iOS [5], and Mixed Reality Toolkit for Microsoft's Universal Windows Platform (UWP) [16]. A-Frame [1] is a library that enables the creation of immersive virtual scenes in the browser by integrating WebVR [32] content within HTML. However, none of these libraries provides specific support for developing and designing visualization applications in XR. Moreover, designing visualizations in immersive environments can be complex, requiring consideration of issues such as occlusion, clutter, and user movement and interactions in 3D [36, 46, 62].

Recent work started to enable easier authoring of immersive visualizations, yet still require a significant amount of low-level programming or are restricted to a limited set of graphical marks. For example, Filonik et al. [53] proposed Glance, a GPU-based framework with a focus on rendering fast and effective abstract visualizations in AR and VR. Donalek et al. [50] developed iViz, which provides a GUI for specifying visualization parameters for a collaborative VR analytics environment. Virtualitics [28] is a commercial immersive and collaborative visualization platform that uses machine learning to help inform the design of three dimensional visualizations. Operations such as filtering, and details-on-demand are supported by virtual pointers.

2.3 Authoring Non-Immersive Visualizations

Visualization authoring tools for non-immersive platforms provide a multitude of approaches, ranging from easy-to-use charting tools to highly flexible visualization programming libraries (Fig. 2).

Fig. 2. Inspired by (top) 2D JavaScript-based authoring tools, (bottom) DXR offers multiple high-level interfaces that are easier to learn and use than low-level Unity programming for constructing visualizations.

For instance, Plotly's Chart Studio [20] lets users interactively explore different styles of charts based on data selection in a tabular view. Similarly, Polestar [21] and RAWGraphs [22] both provide a minimalistic drag-and-drop interface to specify a visual mapping and instantly update the resulting visualization. These interactive charting tools offer easy-to-use graphical interfaces instead of or in addition to programming for adapting pre-configured designs. Tableau [23] combines interactive creation with a scripting interface to perform data analysis. On the other end of the spectrum, there are tools that require more effort to learn and use but allow flexible and novel visualization designs. These tools are naturally closer to low-level programming, but include helper routines such as parsers, color scales, mapping operators, data structures, as well as large libraries of existing visualizations to start with. Examples include D3 for JavaScript [43], the InfoVis Toolkit for Java [52], or Bokeh for Python [7]. Domain-specific languages such as Vivaldi [47], Diderot [59], and ViSlang [64] provide high-level programming APIs that are tailored for application domain experts.

In between these two extremes, there are a set of tools with a trade-off between usability and flexibility. For instance, grammar-based authoring tools provide a high-level abstraction for building visualizations so that designers can focus on their data and not worry about software engineering [55]. The foundational grammar of graphics introduced by Leland Wilkinson [71] paved the way for modern high-level visualization tools such as Vega [67], Vega-Lite [66], and GGplot [70]. Vega and Vega-Lite make visualization design more efficient with concise declarative specifications—enabling rapid exploration of designs albeit with a limited set of graphical marks. Python, R, and Matlab offer their own high-level visualization libraries that require a simple function call with a respective parameterization to deliver data and visualization parameters, e.g., Seaborn, Bokeh, Plotly, GGPlot. Other interactive design tools include Lyra [65], Protovis [42] and Data-Driven Guides [58]. These tools allow for novel designs but require manual specification of shapes, style, and sometimes layout.

DXR integrates several of these approaches. It uses a declarative visualization grammar inspired by Vega-Lite; provides a GUI for specifying visual mappings and designing visualizations inside XR; comes with a set of pre-defined visualizations; and allows for individual styling and customization. DXR is also fully compatible with and can be extended through C# code in Unity.

2.4 Unity Core Concepts

We briefly review the core concepts of Unity as far as they are important for the understanding of DXR. In Unity, applications are represented as composable 3D *scenes* in which designers can add and manipulate *GameObjects* which encapsulate objects and their behavior. Example GameObjects include cameras, 3D models, lights, effects, input handlers, and so on. GameObjects are organized in parent-child *hierarchies* or scene-graphs. GameObjects can be saved as *prefabs* that serve as shareable and reusable templates. Unity has an on-line *Asset Store* [25] for sharing reusable scenes, prefabs, and other assets. A *script* is C# or JavaScript code that can be *attached* to GameObjects as components and used to programmatically manipulate GameObjects at runtime. Designers can edit scenes and GameObjects *interactively* using the Unity Editor user interface, or *programmatically* using scripts via the Unity scripting API [27]. The scene can be configured to run in either AR or VR, simply by specifying the target device in Unity deployment settings. At *runtime*, i.e., when the scene is *played*, the user can see the scene through the device's display, and interact with it using the device's input modalities, e.g., controllers, gesture, or voice. For more information, we refer the reader to the complete Unity documentation [26].

3 DXR Overview

DXR consists of prefabs and scripts that provide a high-level interface for constructing data-driven GameObjects in a Unity scene. Figure 3 illustrates the conceptual parts of DXR. A visualization in DXR is represented as a GameObject prefab—*vis-prefab*—that can be added to scenes and manipulated via the Unity Editor or via scripting, just like any other GameObjects. The vis-prefab reads the visual mapping from a visualization specification file—*vis-specs*—in JSON format. The vis-specs file also contains a URL pointer to the respective data file which can be in CSV or JSON format. When the data or vis-specs file changes, DXR can be notified to update the visual rendering immediately.

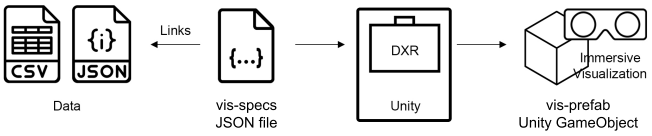

Fig. 3. DXR overview: A *vis-specs* file references the *data* file and holds the design declaration. It gets interpreted by DXR to generate a visualization that is represented as a *vis-prefab* GameObject in Unity.

DXR's main target users are visualization developers with varying expertise in XR programming (Unity/C#) and whose goal is to rapidly prototype and build immersive visualizations.

- **Non-programmers (beginners)** include users with little to no programming experience, e.g., architecture or biology students, artists, and storytellers. In DXR, they can build visualizations without programming. To do this, they can place their data file into the `StreamingAssets/DXRData` directory. Then, they can add a DXR vis-prefab into their scene using the Unity menu or by dragging it into their scene window. They can then set the vis-specs filename parameter of the vis-prefab to an empty file (to start from scratch) or to one of the example vis-specs files in the `StreamingAssets/DXRSpecs` folder containing templates for common visualizations such as bar charts, scatter plots, vector field plots, and many more. At runtime, DXR generates the visualization, and a GUI gives the user control over the data and visual mappings that can be changed (Fig. 4).
- **Non-XR-developers (intermediate)** include users with general programming experience, e.g., with JSON and visualization grammars, but without experience with Unity and C# specifically. With DXR, intermediate users can edit the vis-specs file and directly manipulate the visualization grammar to add or remove visual mappings and fine-tune the design (Sect. 4.1). For example, they can adjust scale domains and ranges, change color schemes, etc. Intermediate users can also create custom graphical marks with generic visual channels without programming (Sect. 6).

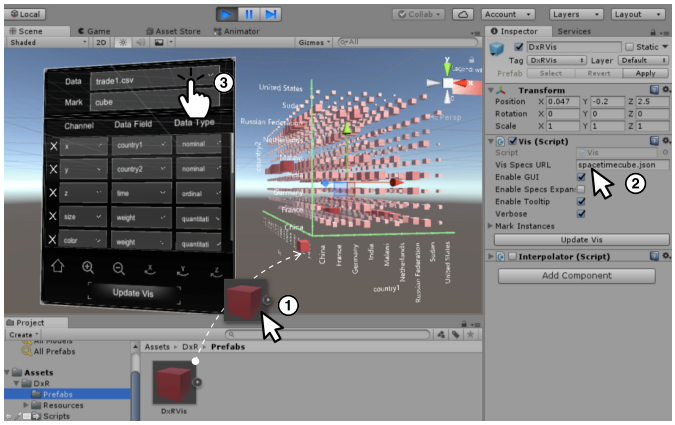

Fig. 4. Steps to use a template: 1) drag-and-drop vis-prefab to the scene, 2) set the vis-specs filename, 3) run, and tweak parameters in the GUI.

- **XR-developers (advanced)** include users experienced in XR programming, i.e., well-versed with Unity/C#'s object-orientedness and scripting API. They are meant to quickly iterate between using the GUI, editing the vis-specs file, and low-level C# programming to develop custom visualizations. Advanced users can build custom visualizations, e.g., by creating a new C# mark class that inherits DXR's graphical mark base class implementation (Sect. 6). GameObject attributes exposed as encoding channels extend the grammar, show up in the GUI in XR, and are available in the vis-specs file. Any new visualization created in this way is now accessible through the previous two scenarios, benefiting other users.

The following sections detail how DXR supports these scenarios.

4 DXR's Visualization Pipeline

Fig. 5 shows DXR's visualization pipeline, consisting of four steps: *specify*, *infer*, *construct*, and *place*. First, the designer describes the visualization design in a concise specification (vis-specs) using DXR's high-level visualization grammar. DXR then infers missing visualization parameters with sensible defaults. Based on this complete specification, DXR then programmatically constructs the 3D visualization that the designer can place in a real or virtual immersive scene.

4.1 Design Specification

We designed DXR's visualization grammar to be similar to Vega-Lite [66] because it is intuitive, making it easier to learn and modify representations, and concise, making it efficient to iterate over designs. Furthermore, there are many visualization designers who are familiar with Vega, Vega-Lite, and Polaris who will find it easier to learn and use DXR and transition their designs to immersive environments.

A single visualization in DXR, which we call `dxrvis`, is a collection of graphical marks (Unity GameObjects) whose properties (position, color, size, etc.) are mapped to data attributes according to the declarative specification in the vis-specs file. Following the notation of Vega-Lite, a `dxrvis` is a simplified equivalent of a *unit* that "describes a single Cartesian plot, with a backing data set, a given mark-type, and a set of one or more encoding definitions for visual channels such as position (x, y), color, size, etc." [66]:

```
dxrvis := (data, mark, encodings, interactions)
```

The input **data** consists of a "relational table consisting of records (rows) with named attributes (columns)" [66]. The **mark** specifies the graphical object (Unity prefab) that will encode each data item. DXR's built-in marks include standard 3D objects like sphere, cube, cone, and text, as well as user-provided custom graphical marks (Sect. 6). The **encodings** describe what and how properties of the mark will be mapped to data. **Interactions** that can be added to the visualization are discussed in a later section. The formal definition of an encoding is:

```
encoding := (channel, field, data-type, value,
scale, guide)
```

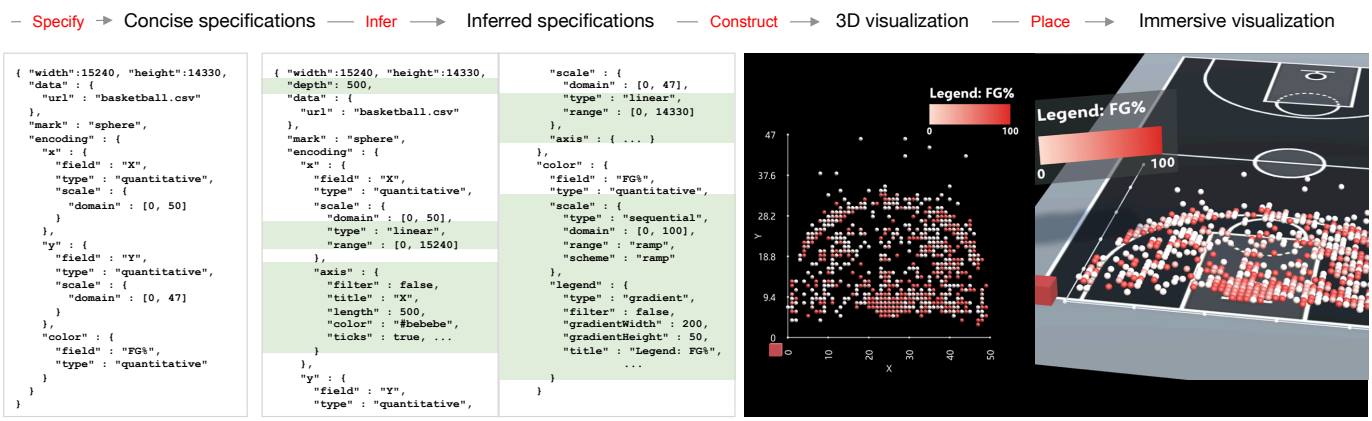

Fig. 5. DXR's visualization pipeline. The designer *specifies* the visualization design via concise specifications. Then DXR *infers* missing parameters to sensible defaults and uses the inferred specifications to programmatically *construct* a 3D visualization that can be *placed* in an AR or VR scene.

The **channel** describes which geometric or visual property of the graphical mark will be mapped to the data attribute specified by the **field**. DXR provides a set of *generic channels* that generally apply to any Unity GameObject, namely `x`, `y`, `z`, `width`, `height`, `depth`, `color`, `opacity`, `size`, `length`, `xrotation`, `yrotation`, `zrotation`, `xdirection`, `ydirection`, and `zdirection`. The `size` channel rescales the object's `width`, `height`, and `depth` equally, while the `length` only rescales the object along a user-defined forward direction. This forward direction is by default set to the (0,1,0) 3D vector, and is used to orient marks that show direction, e.g., arrow and cone. DXR also provides `(x,y,z)offsetpct` channels that translate the mark by a percentage of its `width`, `height`, or `depth` for styling and to handle prefabs with different center or pivot points. The **data-type** describes the data attribute that can be quantitative, nominal, or ordinal. A channel can also be mapped to a fixed setting using the **value** property. The **scale** describes the type of mapping (linear, categorical, etc.) from data attribute values to visual channel properties, as well as the mapping's domain and range. The **guide** properties describe the axis or legend specifications such as tick values, labels, and the like.

Fig. 1 (a-e) and Fig. 5 show examples of declarative specifications using DXR's grammar. A detailed grammar documentation with tutorials is provided on-line at `https://sites.google.com/view/dxr-vis/grammar-docs`. Thanks to the syntax similarity, some Vega-Lite visualizations can be ported with little effort into immersive environments using DXR. Unlike Vega-Lite, DXR does not provide data transforms, yet. We plan to add them in future versions.

4.2 Inference

Concise specifications are intuitive and succinct, making them easy to learn and modify, as well as reduces the tedious setting of *all* tunable visualization parameters. DXR's inference engine sets missing visualization parameters to sensible defaults based on the data types and visual channels informed by the Vega-Lite model. Originally, the resulting inferred specification was hidden from the designer by default and only used internally by DXR. However, feedback from new users indicated that all the possible tunable parameters can be difficult to remember, leading to frequent visits to the on-line documentation. To address this, we provide designers direct access to DXR's inferred specifications so they can see and tweak them directly. This option exposes all tunable parameters to improve debugging, customization, and learning. Inference rules are documented on the DXR website.

4.3 Construction

A DXR specification is not translated to Unity code. Instead, a specification acts as a complete list of parameters for DXR's visualization construction pipeline that gets executed at runtime by the vis-prefab.

Visualizations in DXR are most similar to *glyph*-based visualizations [41]. A graphical mark in DXR is a glyph whose visual properties are mapped to data (independently of other glyphs) and then rendered within a spatial context. Thus, we modeled DXR's construction pipeline after Lie et al.'s *glyph*-based visualization pipeline [61], adapting it to match Unity's scripting API for prefab instantiation and modification. First, DXR parses the data and constructs the necessary internal data structures. Then it loads the specified mark as a GameObject prefab which is instantiated for each data record. Each instance starts with the prefab's default properties with initial positions at the vis-prefab's origin. Then, DXR goes through each encoding parameter in the specifications and changes visual channel properties of the mark instances according to the data attribute or a given fixed value. This instantiation and encoding is performed by a C# *Mark base class* that encapsulates functionalities of a graphical mark. For example, to set the position, rotation, and size channels, the class programmatically modifies each instance's local transform property. Scale parameters in the specification instantiate one of several pre-programmed scaling functions for mapping data attribute values to visual channel values. Finally, DXR constructs optional axes, legends, and query filters. These steps result in an interactive 3D visualization represented as children of the vis-prefab GameObject—a collection of data-driven instances of the mark prefab, with optional axes, legends, and filters. Similar to how glyph instances are rendered in their spatial context, this 3D visualization can be placed in an AR or VR scene for immersion.

We designed this construction pipeline to be as agnostic as possible to the graphical mark prefab's type and complexity in order to support the use of any Unity prefab as graphical mark (Sect. 6).

4.4 Placement

DXR facilitates the placement of visualizations within real or virtual worlds. DXR provides an *anchor*—a red cube near the visualization origin that allows a user to *drag-and-drop* the visualization in a fixed position relative to the real-world or a virtual scene at runtime. When the anchor is *clicked* on, the visualization's position and orientation get attached to that of the user's view. By moving around, the user effectively drags the visualization in 3D space. Clicking on the anchor again drops the visualization. This feature is particularly useful for aligning embedded visualizations with their object referents [72] or spatial contexts such as the examples in Fig. 1. In these embedded visualizations, physical positions of the referents need to be measured and encoded in the data. The anchor can then be aligned to the real-world origin used for measuring these positions. In the future, aligning of graphical marks with (non-)static referents could be enabled with computer vision and tracking.

Furthermore, DXR visualizations are GameObjects that can be composed and placed within a Unity scene either manually using the Unity Editor, or programmatically via scripts, or through libraries such as Vuforia, e.g., for attaching a GameObject to a fiducial marker. In some cases, designers may want to set the size of their visualization to match

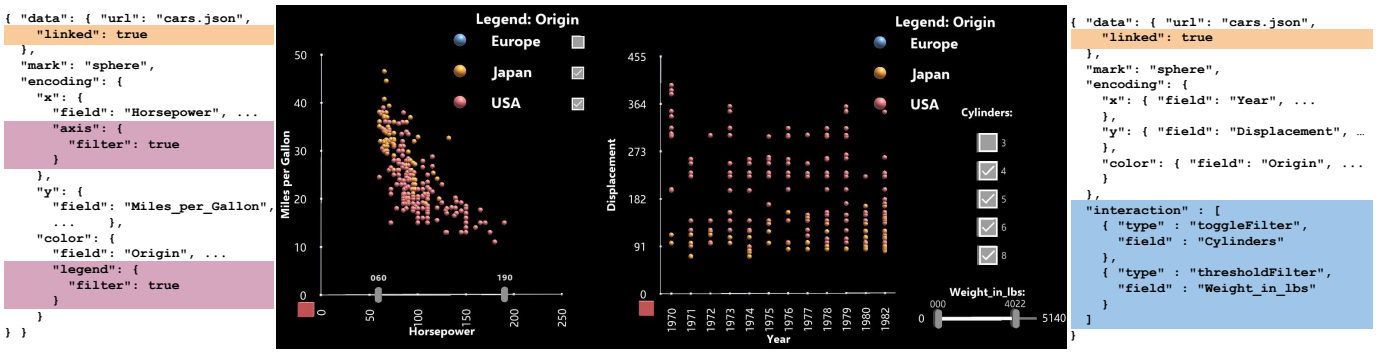

Fig. 6. DXR supports interactive query filters and linked views. For visualized data attributes (e.g., Horsepower, Origin), threshold and toggle filters can be directly integrated into their `axis` and `legend`, respectively (left: purple highlights). For non-visualized attributes (e.g., Cylinders, Weight_in_lbs), filters can be enumerated using the `interaction` parameter (right: blue highlight) and appear on the side. Visualizations that use the same data within the same scene can be linked so that only data items satisfying queries are shown across all linked views (orange highlights).

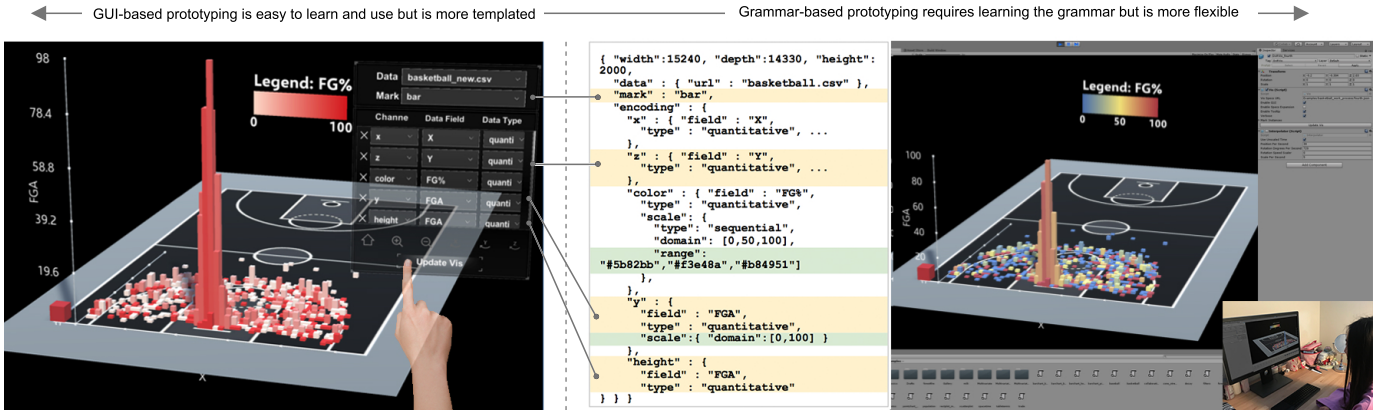

Fig. 7. Prototyping with DXR typically involves (left) trying combinations of data, graphical mark, and visual encoding parameters using the in-situ GUI on an immersive device, and (right) fine-tuning design specifications using a text editor running side-by-side with the Unity Editor on a computer.

the size of their intended canvas when overlaying them together. For example, in Fig. 5, the width and height of the visualization is set to that of the width and height of a basketball court (DXR size units are in millimeters). Moreover, multiple visualizations can be arranged in a scene to enable side-by-side comparison, or building of compound charts, e.g., simple stacked bar charts (Fig. 1h).

4.5 Interactions

In order to support multi-visualization workspaces, DXR allows the creation of query filters and linking of visualizations via vis-specs, illustrated in Fig. 6. Interactive query filters [56] control the visibility of graphical marks according to data using threshold and toggle interfaces. Linked visualizations within the same scene get filtered together.

By default, DXR provides details-on-demand with a tooltip that shows a textual list of data attributes when the user's *gaze* pointer hovers on a graphical mark. DXR's GUI (Sect. 5) also provides view manipulation (scaling up and down, rotating along x, y, z-axis) and view configuration controls. DXR's grammar for interaction can be extended to further exploit device-dependent affordances of tangible [35] or direct manipulation [48] interfaces, as well as gesture- and voice-based input. Furthermore, any existing Unity asset for manipulating GameObjects and navigating scenes can apply to DXR visualizations. For example, hand tracking devices, e.g., leap motion [13], can be utilized to move, rotate, and rescale DXR visualizations using hand gestures. Similarly, device-dependent navigation features such as tracked headsets allow walking around DXR visualizations in AR or VR.

Out of the many options for immersive input modalities, e.g., touch, gaze, gesture, voice [37], we decided to use gaze and click for filtering, GUI interactions, and object placements. This makes them compatible with many common immersive devices because they are typically supported. Their similarity to mouse interactions in WIMP-based interfaces also make them familiar and easy to learn.

5 In-Situ Graphical User Interface (GUI)

Fig. 7 shows a typical XR development set-up. It often requires testing a visualization within an immersive environment while tuning the design on a desktop or laptop computer running the Unity Editor. Initially, we designed DXR so that designers can *only* modify vis-specs in a text editor, typically running side-by-side with the Unity Editor. However, we found that in some cases this led to tedious switching between the two contexts. Furthermore, we found that the JSON syntax and grammar-based specification were overwhelming to non-programmers. To address these challenges, we designed and implemented DXR's in-situ GUI—an intuitive interface that is embedded in the Unity scene with the vis-prefab so it runs *in-situ* within the immersive environment at runtime (Fig. 7:left).

The GUI provides drop-down menus, similar to WIMP interfaces, for selecting data, graphical marks, and visual encoding options from pre-defined sets of parameters. This removes the need to deal with JSON syntax. Moreover, designers no longer need to memorize possible parameter values since the GUI's drop-down menus already provide lists of usable marks, channels, and data attributes. GUI interactions directly modify the underlying specification, as illustrated in Fig. 1 (a, b) and Fig. 7, updating the output visualization instantly for rapid immersive prototyping.

Using the in-situ GUI, a designer can stay in the immersive environment to try different combinations of data, marks, and channels until an initial prototype has been reached. Instant preview of the visualization

gives the designer immediate feedback for rapid design iterations. The design can then be fine-tuned back on the computer using a text editor. The GUI also enables adapting and reusing existing DXR visualizations as pre-configured *templates* similar to interactive charting applications. With only a few clicks in the GUI, an existing visualization's data can be easily changed, instantly updating the visualization with the new data—all without programming or additional scene configuration.

Our initial GUI design included drop-down menus for creating query filters. However, we noticed that in practice they were seldom used, yet made the GUI crowded. In our current design we removed these menus since filters can be easily added via the grammar (Fig. 6). In the future, we plan to make the GUI reconfigurable such that designers can add and arrange menus for features they use the most. Another design we considered was to make the GUI *tag-along* and follow the user's peripheral view. However, multiple GUIs overlap when there are multiple visualizations in a scene, rendering them unusable. In the current design, the GUI is fixed on the side of the visualization by default and simply rotates along the y-axis to always face the user.

6 Custom Graphical Marks and Channels

We made it easy to use any Unity prefab as custom graphical mark in DXR in order to leverage their wide availability and variety to support flexible and engaging visualization designs. Fig. 8 illustrates how DXR enables this by leveraging Unity's object-orientedness in representing graphical marks. As discussed in Sect. 4.3, DXR has a *Mark base class* that encapsulates all mark-related graphing functionalities such as instantiation and visual encoding. This base class treats any mark prefab in the same way, regardless of their type—as a shaded 3D box model. DXR uses the bounding box of this model to modify standard geometric properties like position and size, and its material shader to change color and opacity. This base class is automatically applied to any prefab within a designated *marks directory*.

Any Unity prefab can be converted into a graphical mark in DXR simply by placing it in the marks directory. During construction (Sect. 4.3), DXR uses the `mark` parameter in the specifications as the unique prefab filename to load from the marks directory. Once loaded successfully, the prefab becomes a DXR graphical mark that can be instantiated and modified according to data via the base class implementation. This simple model makes it easy to extend the system with arbitrary Unity prefabs as custom marks. For example, when a 3D model of a book is saved as a prefab in the marks directory, it automatically becomes a DXR graphical mark with the generic channels. Instead of a plain bar chart, this book graphical mark can now be used to generate an embellished bar chart (Fig. 8d).

Optionally, the designer can expose more complex prefab parameters as custom encoding channels by implementing a *derived class* that inherits from DXR's *Mark base class*. For example, using this approach, the intensity property of a flame particle system prefab can be used as an encoding channel, in addition to the generic channels inherited from the base class. This custom mark can be used to visualize forest fires in Montesinho park [17] overlaid on a virtual geographical map (Fig. 8e).

Custom marks and channels are represented as a mark prefab with an optional script of the derived class. These formats can be packed into a Unity *package* file that allows their easy sharing and reuse. Once imported, custom marks and channels just work, without the need for additional set-up or programming. A drawback of this approach however, is that unlike grammars with a fixed set of marks with predictable behavior, DXR users will have to be conscious about imported marks to make sure that they understand how the channel encodings work, to avoid unexpected behavior. In the future, we envision that well documented DXR mark prefabs with accompanying examples will be made available in the Asset Store similar to D3 blocks [8] and Vega or Vega-Lite specifications that will facilitate informed sharing and reuse. Consequently, designers must be conscious when using complex prefabs that could extend construction times or limit frame rates with increasing data size (Sect. 9).

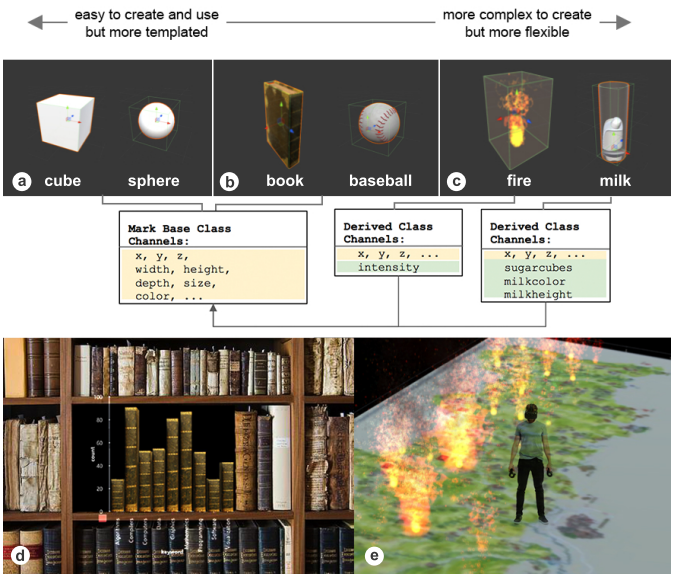

Fig. 8. (a) In addition to DXR's built-in generic graphical marks, (b) designers can use any Unity prefab as a custom mark with generic visual channels such as position, size, and color, simply by saving it in a designated directory. (c) Additional channels can be implemented in a derived class to expose other prefab properties to DXR's visual encoding process. Custom marks and channels enable flexible designs, such as (d) bookshelf keywords visualized using virtual books as bars, and (e) forest fires visualized using flame particle systems on a virtual map.

7 Layered Authoring Support

Studies of visualization design workflows show that designers typically iterate and switch between tools [39, 40]. For example, a designer may use high-level tools like Polaris or Vega-Lite to create an initial visualization, and then switch to more advanced D3 or Vega to fine-tune the design. This type of workflow benefits from *layered authoring* support [54], i.e., cross-compatible tools along the spectrum of simplicity to flexibility illustrated in Fig. 2. This spectrum of tools can also support the collaboration of designers and domain experts with varying expertise in design, visualization, and programming.

DXR naturally supports layered authoring by providing multiple alternative interfaces for design specification and graphical mark customization. For design specification (Fig. 7:top) the GUI is easy to learn and use, but is limited to pre-configured designs since the designer can only change some parameters. The text editor allows tweaking of all tunable parameters but requires familiarity with the grammar and JSON syntax. Similarly, for graphical mark customization (Fig. 8:top), the designer has three options: built-in graphical marks only allow simple designs, custom marks with generic channels are easy to create but only offer standard geometric and visual channels, and custom marks and channels via derived class implementation are more complex to create but are most flexible. With these options, DXR is able to support iterative workflows as well as collaborations among users with varying expertise as illustrated in the following examples.

8 Application Examples

We demonstrate the usage of DXR with a range of representative application examples. Table 1 categorizes them by a variety of characteristics. These and additional examples can be found on the DXR website at `https://sites.google.com/view/dxr-vis/examples`.

Immersive information visualization. DXR can be used to create bar charts (Figs. 1h and 8d), scatter plots (Figs. 1i and 6), and space-time cubes (Fig. 10b) [33]. Without DXR, this would involve writing custom programs to load data, instantiate marks, calculate and apply visual mappings, and create axes, legends, and interactive query filters. With DXR, particularly non-programmers, can easily prototype

Examples	Mark Type		Spatial Dims.		Scale	AR or VR		Anchor	
	Generic	Custom	2D	3D		AR	VR	Real	Virtual
Milk: Fig. 10a		✔	✔		M	✔		✔	
Space-time cubes: Fig. 10b	✔			✔	S, M, L	✔	✔		
VR Workspace: Fig. 10c	✔	✔	✔	✔	M, L		✔		
AR Workspace: Fig. 10d	✔		✔	✔	S	✔		✔	
Buildings: Fig. 10e		✔		✔	S, M, L	✔	✔		✔
Population: Fig. 10f, Flow: Fig. 10g		✔		✔	S, M, L	✔	✔		✔
Streamlines: Fig. 10h		✔		✔	S, M, L	✔	✔	✔	
Baseball: Fig. 1e, Books: Fig. 8d		✔	✔		M	✔		✔	
Photographs: Fig. 1f	✔			✔	M	✔		✔	
Organic Decay: Fig. 1g		✔	✔		L	✔		✔	
Basketball: Fig. 1h	✔		✔		L	✔	✔	✔	✔
Bitcoin: Fig. 1i, Forest Fire: Fig. 8e		✔	✔		S, M, L	✔	✔		✔

Table 1. Summary of examples authored using DXR. **Mark type** is the graphical mark, which can be a generic type (`cube`, `sphere`, `cone`) or a custom prefab; **spatial dimension** is 2D if the visualization uses both `x`, `y` position channels, and 3D if it uses all `x`, `y`, `z` channels; **scale** is the size of the visualization (small: hand size, medium: table size, or large: room size); the runtime environment can be **AR or VR**; and **anchor** specifies whether the visualization is anchored in the real or virtual world.

visualizations without any programming either starting from scratch or reusing templates as previously illustrated in Fig. 4. For example, to visualize research collaborations over time [34], a user can start from scratch with an empty vis-specs file, and then use the GUI to specify the data file, set the visual mark to `cube`, map the categorical attribute `researcher name` to the `x` and `y` channels, map the `time` attribute to the `z` channel, and finally map the quantitative `weights` (collaboration strength) to the cube's `size` and `color`. Based on these parameters, DXR writes the vis-specs file and generates the space-time cube visualization (Fig. 10b:left). This space-time cube can now be used as a *template*. For instance, another user can load a different dataset, e.g., country-to-country trading data, and update the data attributes through the GUI. As the data or any parameters change, the visualization updates (Fig. 10b:right) and the user can proceed with the exploration. A threshold filter for the `weights` attribute can be added using the vis-specs.

Immersive geospatial visualizations. To visualize forest fire data on a virtual map [17], a non-programmer can use a DXR scatter plot template and align its anchor to the origin of a map's image via the Unity Editor (Fig. 9a). At runtime, the user can then use the GUI to set the scatter plot's data filename and assign `xcoord`, `ycoord` of fire location attributes to `x`, `y` channels, and `fire intensity` to `size` (Fig. 9a). An intermediate user might improve on this by creating a custom `flame` graphical mark with generic visual channels. This can be done by downloading a static 3D flame model or prefab, e.g., from an on-line repository, and copy-pasting it in DXR's designated marks directory and renaming it to *flame.prefab*. Using the GUI or by editing the vis-specs file, the graphical mark can then be set to `flame` (Fig. 9b). A more advanced user can use an animated particle system as mark to make the visualization more engaging. To do this, the user can create a derived class, e.g., called *MarkFire*, and override the default `size` channel implementation to map it to the particle system's intensity parameter via C# programming. The user can then directly edit the vis-specs file to set the new mark parameter as well as fine-tune the `range` parameter of the particle system's intensity to match the desired forest fire effect (Fig. 8e).

Similarly, we created a custom 3D bar chart (Fig. 10e) that can be flown-over or walked-through in AR or VR showing heights and ages of several buildings in Manhattan [18]. Furthermore, we downloaded a 3D population visualization [9] from the Asset Store and converted it into a reusable DXR template (Fig. 10f) with minimal effort.

Embedded data visualizations place visualizations close to their physical referent [72]. The example in Fig. 10a embeds a data-driven

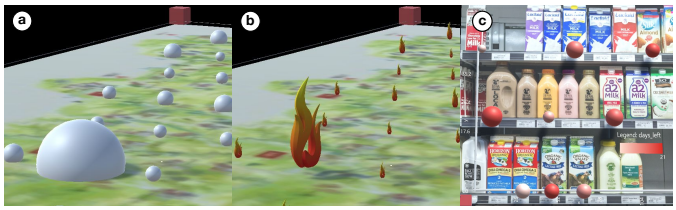

Fig. 9. Prototypes for (a,b) forest fire and (c) milk data visualizations.

virtual glass of milk on top of each milk carton on a grocery shelf. An advanced user would iteratively build this visualization as follows. First, the physical positions of each milk carton are measured (in millimeters, with respect to a real-world origin, e.g., lower left corner of shelf) and are then added as `x` and `y` columns in the data. Then, a 2D scatter plot template is used to visualize the data, using the GUI to map the measured positions to `x`, `y` dimensions, calcium content to `size`, and days before expiration to `color`. Next, the `width` and `height` parameters in the vis-specs are set to match that of the shelf. At runtime, DXR constructs the visualization where scale parameters and color schemes are generated automatically with default values. The user can then place the visualization by aligning its anchor with the shelf's lower left corner (Fig. 9c). Then, the user downloads 3D models of a glass of milk and sugar cubes from Unity's Asset Store and composes a custom graphical mark implementing new channels `milkheight`, `milkcolor`, and `sugarcubes` via C# programming. Using the GUI or vis-specs, these channels are then mapped to calcium content, days to expiry date, and sugar content, respectively. Scale and color schemes are then fine-tuned in the vis-specs, e.g., the color range for the milk is changed from the default white-to-red into brown-to-white reflecting the freshness of the milk (Fig. 10a). For an advanced user, the complete design and specification process can take approximately 15-20 minutes.

Using a custom flame graphical mark's intensity channel, we show the remaining life of referent organic materials hanging on a wall (Fig. 1g), adding a virtual dimension to the existing artwork [3]. And we used DXR to create a 3D vector field plot using built-in cone graphical marks to show locations of photographs (Fig. 1f) of an exhibit [12]. To build this example, 3D real-world positions and orientations were encoded in the data and mapped to the `cone` mark's `x`, `y`, `z`, `xdirection`, `ydirection`, and `zdirection` channels. Embedded data visualizations can reveal insights about physical objects and spaces, as well as enhance our experience in the real-world.

Immersive visualization workspaces consist of multiple linked visualizations in custom AR and VR environments. In an example scenario, users with varying backgrounds can collaboratively develop a VR workspace that visualizes a multi-variate hurricane data [11] through a set of 3D and 2D scientific and abstract visualizations (Fig. 10c). 3D data points encode position, orientation, temperature, pressure, etc. A domain expert on hurricanes without experience in XR-programming, for example, can use a 3D vector field plot template to visualize wind velocity. The GUI can be used to quickly try out different variable combinations to find interesting correlations in the data via scatter plot templates. Working with an advanced immersive visualization designer, the domain expert can then customize the layout, link visualizations, and add query filters to support custom analytical workflows. The arrangement of the visualizations can be easily modified using the Unity Editor or direct manipulation, e.g., such that they surround a physical workstation (Fig. 10d).

Immersive sports data analytics is a growing trend with more and more companies leveraging AR and VR for immersive training and strategizing [30]. Our baseball (Fig. 1e) and basketball (Figs. 1h, 5, and 7) examples were scaled to life-size and blended with real-world or virtual courts. The full immersion potentially makes it easier for players to assimilate and translate the data into actionable insights. With a HoloLens, a baseball batter can, for example, view life-size virtual pitched balls of an opponent for immersive training within a real-world baseball field, similar to an existing immersive training

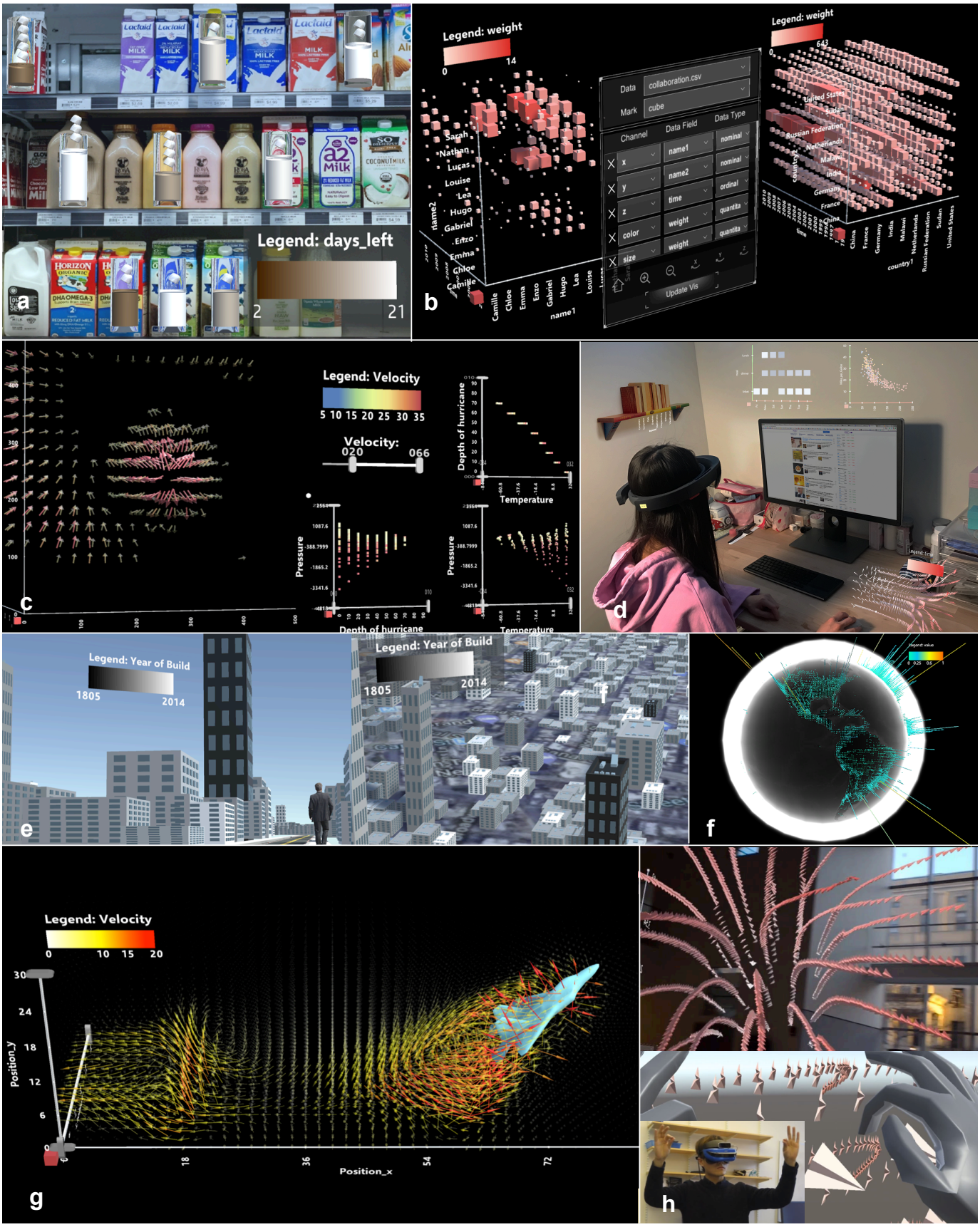

Fig. 10. Examples of immersive visualizations built using DXR include (a) embedded representations, (b, c, d, e, f) 2D and 3D information and geospatial data visualizations, (c, d) immersive workspaces, and (g, h) 3D flow fields and streamlines. Prototyping each example took 10-30 minutes using DXR's GUI and grammar-based interfaces. Custom graphical marks are based on Asset Store prefabs and 3D models from on-line repositories. All examples presented in this paper are available on the DXR website as templates for designers.

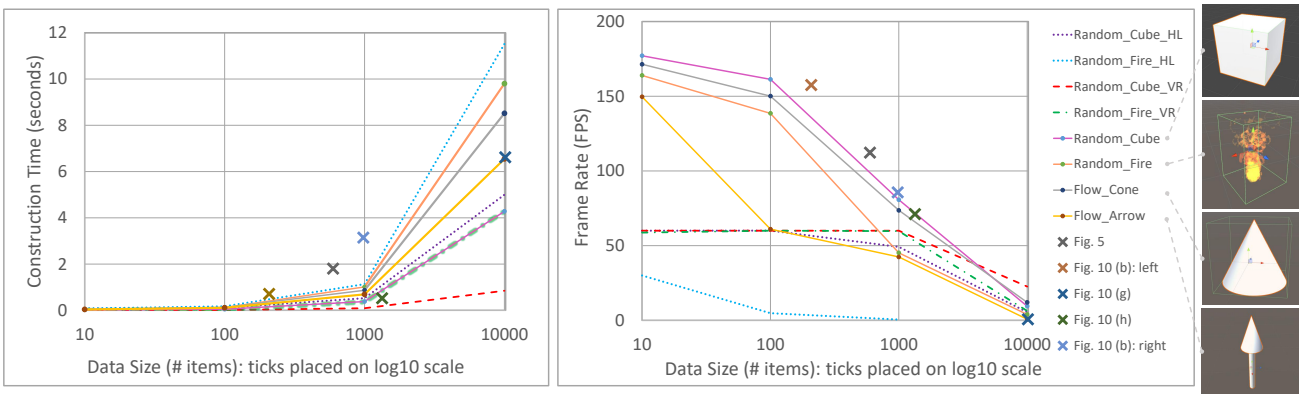

Fig. 11. (left) Construction times and (right) frame rates as a function of data size running on Unity Desktop, HoloLens (HL), and ACER VR headset (VR). Lines show visualizations that use simple (cube, cone) or complex (fire, arrow) graphical marks. **X** marks show other examples from this paper.

application [30]. Since most sports data are similar across players and teams, once their visualizations are implemented in DXR, they can be reused as templates by non-programmer players and coaches.

We also used DXR to create immersive flow field (Fig. 10g) and streamlines (Fig. 10h) visualizations using arrow and paper airplane graphical marks, respectively. Fig. 10h shows direct manipulation of DXR visualizations using a leap motion controller.

9 Performance Evaluation

As DXR is meant for prototyping and exploring designs, scalability was not an explicit design goal. This section reports on performance measures of the current implementation. Fig. 11 shows construction times and frame rates for varying data sizes and graphical mark complexities running on Unity Desktop, HoloLens [15] (HL), and ACER VR headset [2] (VR). The Unity Desktop experiments were performed using Unity Editor 2017.2.1 on a PC with an Intel Xeon CPU (2 processors @ 2.10 GHz), 128 GB RAM, and a GTX Titan X graphics card. The ACER VR headset was tethered to the same PC, while the HoloLens used its own standalone processor and GPU. For the `Random_Cube` and `Random_Fire` examples, we generated random 3D points and plotted them in a 3D scatter plot. We used the built-in cube graphical mark and more complex fire particle system similar to Fig. 8e, respectively. We used these two examples as representative visualization examples with both simple and complex marks on all devices. The `Flow_Cone` and `Flow_Arrow` examples use the flow simulation data shown in Fig. 10g at different subsampling levels plotted as a vector field. We used the built-in cone and custom arrow graphical marks, respectively. Note that the flow visualization examples used 8 channels (`x`, `y`, `z`, `color`, `opacity`, `xdirection`, `ydirection`, `zdirection`), while the scatter plot used only 3 channels (`x`, `y`, `z`).

To measure **construction time**, we ran DXR's visualization construction pipeline (Sect. 4) 11 times for each example. We discarded the first one as warm-up and report the average of the remaining 10. Construction times remain below 12 seconds even for complex examples. As data size goes up, construction times increase as an effect of increasing graphical mark prefab instantiation calls.

When measuring **frame rate**, we kept the visualization fully visible within the viewport and continuously rotated it along the y-axis with respect to its center. Frame rates drop more or less exponentially with increasing complexity of the scene. Scenes with 10,000 items are barely interactive (60 FPS or less [19]). The `Flow_Arrow` example (yellow line) drops quickly because our custom arrow graphical mark consists of two geometric primitives, a cone and cylinder, that need to be rendered and mapped to 8 data attributes each.

For reasonable data complexity (1,000 items or less) the system achieves real-time frame rates (over 50 FPS). The exception to this is the HoloLens which performs poorly with highly complex mark prefabs, e.g., fire particle system, due to its limited stand-alone GPU inside the head-mounted display. Nevertheless, for applications running below 60 FPS, the HoloLens runs a built-in image stabilization pipeline that improves the stability of virtual objects to reduce motion sickness, e.g., by duplicating frames [10]. We are not able to run the `Random_Fire_HL` example with 10,000 data points on the HoloLens due to memory limitations. We also note that the HoloLens and the VR headset automatically cap frame rates at 60 FPS. With these numbers in mind, designers must be conscious about their choice of mark prefab in order to balance prefab complexity, data size, and frame rates according to their design and hardware requirements. For data sizes beyond 1,000 items, despite low frame rates, DXR can still benefit developers in *quickly and cheaply previewing* visualization designs in XR *before* investing time in writing specialized and optimized implementations.

In the future, we hope to leverage advanced GPU shader programs to improve frame rates for large data sizes. We could also provide specially optimized custom graphical marks and use level-of-detail techniques that have been developed to handle large-scale scientific visualizations [63]. Eventually, designers can build on DXR to implement more scalable custom visualization techniques, e.g., multi-resolution approaches, by aggregating data via external tools, combining multiple visualizations, and customizing mark behavior.

10 Conclusions and Future Work

DXR makes rapid prototyping of immersive visualizations in Unity more accessible to a wide range of users. By providing a high-level interface and declarative visualization grammar, DXR reduces the need for tedious manual visual encoding and low-level programming to create immersive data-driven content. We believe DXR is an important step towards enabling users to make their data engaging and insightful in immersive environments.

DXR opens up many directions for future work. On one hand, we look forward to developing new immersive visualization applications for shopping, library-browsing, office productivity systems, or collaborative analysis. On the other hand, we encourage the user community to improve and extend DXR's functionality. In addition to the GUI, alternative immersive interfaces can be explored for specifying and interacting with data representations, e.g., using gesture, voice, or tangible user interfaces. We envision the development of immersive visualization recommender systems, similar to Voyager, providing better support for designing in the AR-CANVAS [36] and to suggest designs that can alleviate potential cluttering and occlusion issues. DXR may also enable perception and visualization researchers to streamline user studies for a better understanding of the benefits and limitations of immersive visualization in various domains.

Acknowledgments

The authors wish to thank Iqbal Rosiadi, Hendrik Strobelt, and the anonymous reviewers for their helpful feedback and suggestions. This work was supported in part by the following grants: NSF IIS-1447344, NIH U01CA200059, and National Research Foundation of Korea grants NRF-2017M3C7A1047904 and NRF-2017R1D1A1A09000841.

REFERENCES

[1] A-Frame. https://aframe.io/. Last accessed: March 2018.

[2] Acer Mixed Reality. https://www.acer.com/ac/en/US/content/series/wmr. Last accessed: March 2018.

[3] Aguahoja. http://www.cpnas.org/exhibitions/current-exhibitions/aguahoja.html. Last accessed: July 2018.

[4] ARCore. www.developers.google.com/ar/discover/. Last accessed: March 2018.

[5] ARKit. www.developer.apple.com/arkit. Last accessed: March 2018.

[6] ARToolkit. www.artoolkit.org. Last accessed: March 2018.

[7] Bokeh. https://bokeh.pydata.org/en/latest. Last accessed: March 2018.

[8] D3 Blocks. https://bl.ocks.org/mbostock. Last accessed: March 2018.

[9] Globe - Data Visualizer. https://assetstore.unity.com/packages/templates/systems/globe-data-visualizer-80008. Last accessed: March 2018.

[10] Hologram stability. https://docs.microsoft.com/en-us/windows/mixed-reality/hologram-stability. Last accessed: June 2018.

[11] IEEE Visualization 2004 Contest. http://sciviscontest-staging.ieeevis.org/2004/data.html. Last accessed: March 2018.

[12] Jerome B. Wiesner: Visionary, Statesman, Humanist. https://www.media.mit.edu/events/jerome-b-wiesner-visionary-statesman-humanist/. Last accessed: July 2018.

[13] Leap Motion. https://www.leapmotion.com/. Last accessed: March 2018.

[14] LookVR. https://looker.com/platform/blocks/embedded/lookvr. Last accessed: March 2018.

[15] Microsoft HoloLens. https://www.microsoft.com/en-us/hololens. Last accessed: March 2018.

[16] Mixed Reality Toolkit. https://github.com/Microsoft/MixedRealityToolkit-Unity.

[17] Montesinho Park Forest Fires Data. https://www.kaggle.com/elikplim/forest-fires-data-set. Last accessed: March 2018.

[18] New York City Buildings Database. https://www.kaggle.com/new-york-city/nyc-buildings/data. Last accessed: March 2018.

[19] Performance Recommendations for Hololens Apps. https://docs.microsoft.com/en-us/windows/mixed-reality/performance-recommendations-for-hololens-apps. Last accessed: March 2018.

[20] Plotly. www.plot.ly. Last accessed: March 2018.

[21] Polestar. http://vega.github.io/polestar/. Last accessed: March 2018.

[22] RAWGraphs. https://rawgraphs.io/. Last accessed: March 2018.

[23] Tableau. https://www.tableau.com/. Last accessed: March 2018.

[24] Unity. www.unity3d.com. Last accessed: March 2018.

[25] Unity Asset Store. https://assetstore.unity.com/. Last accessed: March 2018.

[26] Unity documentation. https://docs.unity3d.com/Manual. Last accessed: June 2018.

[27] Unity Scripting API. https://docs.unity3d.com/ScriptReference/. Last accessed: March 2018.

[28] Virtualitics. https://www.virtualitics.com/. Last accessed: March 2018.

[29] Visualization Toolkit. https://www.vtk.org/. Last accessed: March 2018.

[30] VR Sports Training. https://www.eonreality.com/portfolio-items/vr-sports-training/. Last accessed: March 2018.

[31] Vuforia. https://www.vuforia.com/. Last accessed: March 2018.

[32] WebVR. https://webvr.info. Last accessed: March 2018.

[33] B. Bach, P. Dragicevic, D. Archambault, C. Hurter, and S. Carpendale. A descriptive framework for temporal data visualizations based on generalized spacetime cubes. *Computer Graphics Forum*, 36(6):36–61, 2017. doi: 10.1111/cgf.12804

[34] B. Bach, E. Pietriga, and J.-D. Fekete. Visualizing dynamic networks with matrix cubes. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pp. 877–886. ACM, 2014.

[35] B. Bach, R. Sicat, J. Beyer, M. Cordeil, and H. Pfister. The hologram in my hand: How effective is interactive exploration of 3D visualizations in immersive tangible augmented reality? *IEEE Transactions on Visualization and Computer Graphics*, 24(1):457–467, Jan 2018. doi: 10.1109/TVCG.2017.2745941

[36] B. Bach, R. Sicat, H. Pfister, and A. Quigley. Drawing into the AR-CANVAS: Designing embedded visualizations for augmented reality. In *Workshop on Immersive Analytics, IEEE Vis*, 2017.

[37] S. K. Badam, A. Srinivasan, N. Elmqvist, and J. Stasko. Affordances of input modalities for visual data exploration in immersive environments. In *Workshop on Immersive Analytics, IEEE Vis*, 2017.

[38] M. Bellgardt, S. Gebhardt, B. Hentschel, and T. Kuhlen. Gistualizer: An immersive glyph for multidimensional datapoints. In *Workshop on Immersive Analytics, IEEE Vis*, 2017.

[39] A. Bigelow, S. Drucker, D. Fisher, and M. Meyer. Reflections on how designers design with data. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*, AVI '14, pp. 17–24. ACM, New York, NY, USA, 2014. doi: 10.1145/2598153.2598175

[40] A. Bigelow, S. Drucker, D. Fisher, and M. Meyer. Iterating between tools to create and edit visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):481–490, Jan 2017. doi: 10.1109/TVCG.2016.2598609

[41] R. Borgo, J. Kehrer, D. H. Chung, E. Maguire, R. S. Laramee, H. Hauser, M. Ward, and M. Chen. Glyph-based visualization: Foundations, design guidelines, techniques and applications. In *Eurographics (STARs)*, pp. 39–63, 2013.

[42] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, Nov. 2009. doi: 10.1109/TVCG.2009.174

[43] M. Bostock, V. Ogievetsky, and J. Heer. Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec 2011. doi: 10.1109/TVCG.2011.185

[44] S. Butscher, S. Hubenschmid, J. Müller, J. Fuchs, and H. Reiterer. Clusters, trends, and outliers: How immersive technologies can facilitate the collaborative analysis of multidimensional data. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pp. 90:1–90:12. ACM, New York, NY, USA, 2018. doi: 10.1145/3173574.3173664

[45] T. Chandler, M. Cordeil, T. Czauderna, T. Dwyer, J. Glowacki, C. Goncu, M. Klapperstueck, K. Klein, K. Marriott, F. Schreiber, and E. Wilson. Immersive Analytics. In *2015 Big Data Visual Analytics (BDVA)*, pp. 1–8, Sept 2015. doi: 10.1109/BDVA.2015.7314296

[46] Z. Chen, Y. Wang, T. Sun, X. Gao, W. Chen, Z. Pan, H. Qu, and Y. Wu. Exploring the design space of immersive urban analytics. *Visual Informatics*, 1(2):132 – 142, 2017. doi: 10.1016/j.visinf.2017.11.002

[47] H. Choi, W. Choi, T. M. Quan, D. G. C. Hildebrand, H. Pfister, and W. K. Jeong. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2407–2416, Dec 2014. doi: 10.1109/TVCG.2014.2346322

[48] M. Cordeil, A. Cunningham, T. Dwyer, B. H. Thomas, and K. Marriott. ImAxes: Immersive axes as embodied affordances for interactive multivariate data visualisation. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pp. 71–83. ACM, New York, NY, USA, 2017. doi: 10.1145/3126594.3126613

[49] M. Cordeil, T. Dwyer, K. Klein, B. Laha, K. Marriott, and B. H. Thomas. Immersive collaborative analysis of network connectivity: Cave-style or head-mounted display? *IEEE Transactions on Visualization and Computer Graphics*, 23(1):441–450, Jan 2017. doi: 10.1109/TVCG.2016.2599107

[50] C. Donalek, S. G. Djorgovski, S. Davidoff, A. Cioc, A. Wang, G. Longo, J. S. Norris, J. Zhang, E. Lawler, S. Yeh, A. Mahabal, M. J. Graham, and A. J. Drake. Immersive and collaborative data visualization using virtual reality platforms. In *Big Data (Big Data), 2014 IEEE International Conference on*, pp. 609–614. IEEE, 2014.

[51] N. ElSayed, B. Thomas, K. Marriott, J. Piantadosi, and R. Smith. Situated analytics. In *2015 Big Data Visual Analytics (BDVA)*, pp. 1–8, Sept 2015. doi: 10.1109/BDVA.2015.7314302

[52] J. D. Fekete. The InfoVis Toolkit. In *IEEE Symposium on Information Visualization*, pp. 167–174, 2004. doi: 10.1109/INFVIS.2004.64

[53] D. Filonik, T. Bednarz, M. Rittenbruch, and M. Foth. Glance: Generalized geometric primitives and transformations for information visualization in AR/VR environments. In *Proceedings of the 15th ACM SIGGRAPH Conference on Virtual-Reality Continuum and Its Applications in Industry*

- *Volume 1*, VRCAI '16, pp. 461–468. ACM, New York, NY, USA, 2016. doi: 10.1145/3013971.3014006

[54] M. Gandy and B. MacIntyre. Designer's augmented reality toolkit, ten years later: Implications for new media authoring tools. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pp. 627–636. ACM, New York, NY, USA, 2014. doi: 10.1145/2642918.2647369

[55] J. Heer and M. Bostock. Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1149–1156, Nov 2010. doi: 10.1109/TVCG.2010.144

[56] J. Heer and B. Shneiderman. Interactive dynamics for visual analysis. *Queue*, 10(2):30:30–30:55, Feb. 2012. doi: 10.1145/2133416.2146416

[57] R. Kenny and A. A. Becker. Is the Nasdaq in another bubble? A virtual reality guided tour of 21 years of the Nasdaq. `http://graphics.wsj.com/3d-nasdaq/`. Last accessed: March 2018.

[58] N. W. Kim, E. Schweickart, Z. Liu, M. Dontcheva, W. Li, J. Popovic, and H. Pfister. Data-Driven Guides: Supporting expressive design for information graphics. *IEEE Transactions on Visualization and Computer Graphics*, PP(99):1–1, Jan 2017 2017.

[59] G. Kindlmann, C. Chiw, N. Seltzer, L. Samuels, and J. Reppy. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):867–876, Jan 2016.

[60] J. Li. Beach. `https://www.jiabaoli.org/#/beach/`. Last accessed: March 2018.

[61] A. E. Lie, J. Kehrer, and H. Hauser. Critical design and realization aspects of glyph-based 3D data visualization. In *Proceedings of the 25th Spring Conference on Computer Graphics*, SCCG '09, pp. 19–26. ACM, New York, NY, USA, 2009. doi: 10.1145/1980462.1980470

[62] M. Luboschik, P. Berger, and O. Staadt. On spatial perception issues in augmented reality based immersive analytics. In *Proceedings of the 2016 ACM Companion on Interactive Surfaces and Spaces*, ISS Companion '16, pp. 47–53. ACM, New York, NY, USA, 2016. doi: 10.1145/3009939.3009947

[63] Z. Lv, A. Tek, F. Da Silva, C. Empereur-mot, M. Chavent, and M. Baaden. Game on, Science - How video game technology may help biologists tackle visualization challenges. *PLOS ONE*, 8(3):1–13, 03 2013. doi: 10.1371/journal.pone.0057990

[64] P. Rautek, S. Bruckner, M. E. Gröller, and M. Hadwiger. ViSlang: A system for interpreted domain-specific languages for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2388–2396, Dec 2014.

[65] A. Satyanarayan and J. Heer. Lyra: An interactive visualization design environment. In *Computer Graphics Forum*, vol. 33, pp. 351–360. Wiley Online Library, 2014.

[66] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, Jan 2017. doi: 10.1109/TVCG.2016.2599030

[67] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, Jan 2016.

[68] W. Usher, P. Klacansky, F. Federer, P. T. Bremer, A. Knoll, J. Yarch, A. Angelucci, and V. Pascucci. A virtual reality visualization tool for neuron tracing. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):994–1003, Jan 2018. doi: 10.1109/TVCG.2017.2744079

[69] S. White and S. Feiner. SiteLens: Situated visualization techniques for urban site visits. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pp. 1117–1120. ACM, New York, NY, USA, 2009. doi: 10.1145/1518701.1518871

[70] H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer, 2016.

[71] L. Wilkinson. *The grammar of graphics*. Springer Science & Business Media, 2006.

[72] W. Willett, Y. Jansen, and P. Dragicevic. Embedded data representations. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):461–470, Jan 2017. doi: 10.1109/TVCG.2016.2598608