D3GL: A D3 Plug-in for 3D Data Visualization Using WebGL

Jiwon Kim
jiwonk@cs.stanford.edu

Daniel Posch
dcposch@cs.stanford.edu

ABSTRACT

In this paper we present a D3 plug-in library that lets users control WebGL elements and bind data in an intuitive way. WebGL is a powerful but unwieldy Javascript library, the only way to do hardware-accelerated 3D rendering in the browser. It is a thin wrapper on a low-level C API called OpenGL ES. D3GL abstracts away the setup and provides the user with a streamlined interface to render and manipulate data visualizations. It currently supports interactions including zoom, rotation, and selection with various data primitives such as points, user-defined overlays, shapes, and bars on globes.

Author Keywords

Data Visualization, JavaScript, D3, WebGL, 3D Visualization, Spatial Data

INTRODUCTION

There are several types of data that would benefit from a three-dimensional visualization. One example is global, location-based data. 2D projections of the globe pose problems of distortion and misrepresentation. To take a simple, common example, a Mercator projection of the world's countries has a large "lie factor", because countries like Greenland and Russia appear much larger than they actually are. 3D visualization has a lie factor of one. Through projection, rotation, zooming, etc, 3D visualization leverages the human brain's powerful faculties for understanding spatial data. Flight paths, for instance, make much more sense when depicted on a three-dimensional Earth because the best paths fit the curve of the Earth, which may look strange and arbitrary on a flat projection of the world. Stellar bodies other than the Earth may also benefit from three- dimensional visualizations, particularly because unlike the world map, most viewers are not familiar with flat maps of the sun, moon, Mars, etc. Other examples of spatial data that will benefit from three-dimensional representations include point-cloud data, surface data for arbitrary shapes, CAD (computer-aided design) iterations for three-dimensional objects.

However, we have found no user-friendly libraries or frameworks for the purpose of versatile 3D data visualization. D3GL makes data visualization with WebGL an elegant process by adopting D3's style of binding views to the underlying data. There is an important difference, however—D3GL must contend with lower-level primitives. D3 binds data directly to SVG and HTML primitives, such as <circle> and . It is not nearly as useful to bind data to WebGL primitives such as triangles and float buffers. To solve this, we've created higher-level 3D primitives. Instead of offering the user a direct manipulation of vertices and triangles, which may be daunting, D3GL presents the user with a globe with data element primitives such as points, user-defined texture overlays, shapes, bars, and arcs for display of data, as well as interaction with the elements and the globe including zoom, rotation, and selection. In the future, D3GL will include more primitives such as user-provided meshes and point clouds.

DESIGN CONSIDERATIONS

Because of the myriad of different things WebGL could accomplish and the many types of three-dimensional visualizations that could be realized, we needed to scope the project and hone its focus. Some types of data do not benefit from a three-dimensional view. Portraying a simple bar chart in 3D, for example, would be a waste of ink. Rather than use WebGL's capabilities as our starting point, we've thought about the types of data for which a third dimension adds the most value.

We decided to start out with a 3D globe. This was after we observed that successful data visualizations in 3D were likely to be spatial data directly linked to the real world. In deciding what types of data primitives to provide so that the user can overlay the data over the 3D globe, we thought of the possible use cases the user might find for D3GL.

Points would be useful for indicating locations on the globe. With the freedom that the user has to determine the texture of the globe and the color and radius of each point, points can be used to represent anything from landings on Mars to population densities across the world. Since points support selection, the developer also has the freedom to respond to the

selection of a point in any way, like showing details or a linked selection in another visualization.

Arbitrary overlays provide the user with absolute freedom to draw anything on the globe, using the HTML5 canvas 2D drawing API. D3GL provides utilities to make this simple.

Furthermore, we wanted to support color-coded maps, which support interaction with the individual regions. We started with a set of built-in regions, such as the countries of the world. During the process of developing choropleth maps, we realized that we should support developer-defined shapes for domains that disregard country borders, such as ethnic distribution, wildlife populations, and more.

Bars would be a 3D, geo-spatial embodiment of bar charts. Not like the 3D bar charts popularized by MS Office, which redundantly show two-dimensional data, but rather charts where each bar has a location in space, showing three or more data dimensions. The decision to support bars was influenced by the different demos we saw of the Chrome WebGL Globe demo, in which different datasets such as server activities and world population were visualized with bars jutting radially out of the earth, sometimes color -coded to portray an additional dimension of data.

Lastly, we provide arcs, a 3D primitive that allows developers to connect two points on a globe. This is useful for visualizing trade flows, airplane flights, and many other types of data.

RELATED WORK

As far as we know, D3GL is the only open-source library for three-dimensional data visualization in the browser. There are libraries for 2D data visualization in the browser, such as D3. There are libraries on top of WebGL, such as Three.JS, but these are pure graphics libraries, and don't provide a way to work with data. Finally, there are 3D data visualization products with desktop clients, such as Ayasdi. We believe D3GL is the first Javascript library for hardware-accelerated 3D data visualization.

During our research, we identified roughly three categories of existing 3D data visualization tools.

2D Visualization with 3D for aesthetics

These programs generally encourage the non-programmer to provide data, which it automatically renders in 3D. Such programs include Microsoft Excel, Microsoft PowerPoint, and DataAppeal.

DataAppeal is a for-profit web-based mapping application "designed to visualize geospatial data using captivating models and images." It overlays three-dimensional elements such as spikes and spheres over Google Earth to visualize data. Although the idea resonates with ours, the demos for DataAppeal demonstrate a lack of understanding of which visualization suits what type of data, from the fact that sphere volumes are recommended to represent value, when in fact the nature of the inaccuracies of human perception of volume and the depth perspective of Google Earth make it difficult to portray and understand data.

Figure 2: Toronto Population Demo for DataAppeal

Demos

This category includes beautiful three-dimensional data visualizations that are one-offs, written directly on top of low-level APIs like OpenGL or WebGL.

WebGL Globe by Chrome lets the user bind data to a 3D Earth that displays bars jutting radially out of the globe in different colors according to the value of the data element. The globe features smooth zoom, rotation, transition between datasets, and a beautiful atmospheric effect. The existing examples that visualize datasets such as world population and incomes give the viewer a lasting impression on the general trend. Interaction, however, is limited to zoom and rotation. Thus it is impossible to look up the value of any data element from its bar representation. Moreover, it is not an API—in order to make your own WebGL Globe, you are encouraged to "copy the code, add your own data."

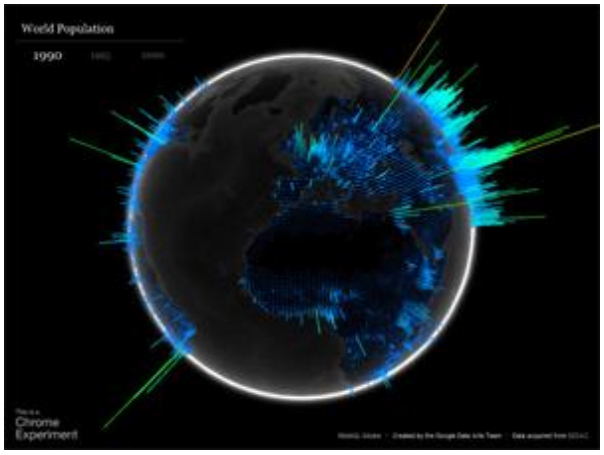


Figure 1: World Population Demo for WebGL Globe by Chrome

WebGL Frameworks

There exist several libraries that provide abstractions for WebGL, as D3GL does. These focus on 3D rendering capability, such as for gaming, not on visualizing data. In providing the user with the ability for a more fine-grained control over WebGL elements such as cameras, vertices, and matrices, most libraries are still burdensome to use for those who are looking for an efficient way to bind and visualize spatial data.

PhiloGL is a WebGL framework for "data visualization, creative coding and game development." The demos "World Airline Routes" and "Temperature Anomalies from 1880 to 2010" visualize worldwide location-based data on an interactive globe, the airline routes demo having been an inspiration for the arcs data primitive for D3GL. PhiloGL lacks the simplicity that could be achieved by abstracting away most of the logic around cameras, individual vertices, and matrix transformations.

Figure 3: World Flights Demo for PhiloGL

METHODS

We created a library of several data primitives and interaction techniques. The goal is to allow developers to easily bind data to these primitives to make spatial visualizations.

Creating Globes

To add a globe, you first create a globe template and set its properties using chaining, as in D3.js:

```
var globe = d3.gl.globe()
        .width(300)
        .height(300)
        .texture(function(d) {
                return "../"+d+"-tex.jpg";
                // eg ../earth-tex.jpg
        });
```

Figure 4: Creating a globe template

The globe template serves as a rendering template for multiple globes. To configure a specific property per globe, a function is passed in as argument, as in the `texture` call above. For properties shared among all globes, a static value can be passed in, as in the `width` and `height` calls.

The globe(s) can be rendered by binding data, appending DOM elements to contain them, and calling globe, which refers to the object previously initialized with a call to `d3.gl.globe()`:

```
var data = ['earth', 'moon'];
d3.select("body").selectAll("span")
        .data(data)
        .enter()
        .append("span")
        .call(globe);
```

Figure 5: Rendering two globes with bound data

This is analogous to the D3.js style for adding elements bound to data. The above code will pass the bound data, which is the array `data`, when it calls globe.

Then for each element in the data that is passed in, a new D3 globe object is created to be rendered on screen. Each element in the data array will be passed in as argument. In this example, the first globe is passed in `'earth'` and the second `'moon'`, which are used to determine the texture of each globe.

The conventions for this API were inspired by Michael

Bostock's essay "Toward Reusable Charts," which encourages the use of closures with getter and setter methods and outlines specific design patterns. Thus `d3.gl.globe` is a closure with properties such as width, height, texture, transparency, zoom, and rotation. It supports method chaining.

Internal Implementation

When `call` is invoked on `globe` in Figure 5, a function is called for each data element in the bound data. The appropriate values for the properties are set using user-provided values and functions. Then WebGL is initialized using Three.js, and a render function is called and scheduled to be called 60 frames per second using RequestAnimationFrame.

Binding Data to Each Globe

So far we have discussed binding a dataset to a globe-rendering template, then creating multiple globes that utilize the template. This is how you bind a dataset to each globe, rather than to the globe template:

```
var rainfall = [10, 0, 5, 20];
globe.points().data(rainfall);
```

Figure 6: Binding a dataset to each globe

Figure 6 binds the dataset `rainfall` to the globes. Before the data is bound, `points` is called. This is because data is actually bound not only to each globe, but to each overlay on each globe. Thus one globe can have multiple overlays that are bound to different datasets.

But Figure 6 passes in the same dataset to all the globes that are created. In order for the different globes to get different datasets, we can do the following:

```
var precipitation = {
        'rainfall': [10, 0, 5, 20],
        'snowfall': [0, 30, 0, 10]
};
// create globe rendering template
var globe = d3.gl.globe()
    .width(300).height(300);
// add points, specify dataset for points
var points = globe.points().data(
    function(d) {return precipitation[d];});
d3.select("body").selectAll("span")
    .data(Object.keys(precipitation))
    .enter()
    .append("span")
    .call(globe);
```

Figure 7: Binding different datasets per globe

In Figure 7, each data element from `precipitation` is passed in per globe, which is then used to get the right dataset.

Selection

One of the most intuitive ways to interact with a visualization of data is to select individual data elements to reveal further information. In the related works, none presented a simple way for the user to bind mouse events to relevant data elements. With the `.on` mouse handlers D3GL makes available, the user can access individual data elements.

The mouse event can be linked to an overlay type as the following:

```
var mouseoverPoint;
points.on("mousemove", function(evt) {
        mouseoverPoint = evt.point;
});
```

Figure 8: Adding mouse events to an overlay

In the above snippet of code the variable `points` is defined as in the previous section. When the mouse moves on the globe, the user-defined function is invoked with the mouse event as argument. The data element on the globe that the mouse is pointing to is stored under `evt.point`, and Figure 8 assigns the data element to the variable mouseoverPoint. `evt.point` is `null` if there are no data elements under the mouse.

D3GL supports the events `mousemove`, `mousedown`, `mouseup`, `click`, and `dblclick`. Different overlays store data elements under different names – `evt.point`, `evt.shape`, `evt.bar`, etc. You can bind events separately for points, bars, etc.

Internal Implementation

To implement mouse interaction, we layer the mouse event for each overlay on top of the mouse event for the globe. When each globe is initialized, the globe starts listening to any mouse events on the `canvas` element that contains it. The globe also stores a list of handlers that the user specifies for each event, such as when the user calls `points.on` in Figure 8. When a listener picks up on an event, an intersect method is called the figures out the latitude and longitude of the point on the globe under the mouse. Ray tracing is used to obtain this information. Then the computed latitude and longitude are stored in the mouse event

object, which is passed into all the handlers for the mouse event that are fired subsequently.

For example, when the user calls `points.on`, the points closure creates a function that takes a mouse event object evt, whose properties include evt.lat and evt.lon, the latitude and longitude computed when the mouse event is caught by the handlers in the globe closure. Then this function uses this information to find the exact data element that it corresponds to. Finally, after storing this information in evt.point (or evt.bar, evt.shape, etc), the handler created by the overlay invokes the user's callback.

This layered approach to handling mouse events provides different layers of abstraction: The globe does not assume anything about mouse handlers that it stores – it just fires them with a computed latitude and longitude. The overlays use the latitude and longitude computed to find the data element, using procedures specific to the overlay type. The client-specified callback uses the stored data element to respond to the mouse event, without having to worry about lower-level details such as the bubbling of mouse events from the canvas, to the globe, to the overlay.

Points

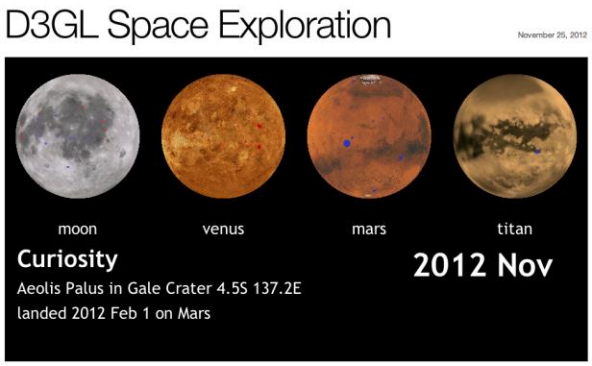


Figure 9: Points specify landings on stellar bodies in a D3GL demo

Points are the simplest way to show location data. By letting the user define the placement, color, and size of points, several data dimensions can be visualized. The following example places each point at the location indicated by the data element's latitude and longitude properties, color that varies between white and blue according to the data element's value, and a fixed radius of 0.1 degrees on the globe:

```
var c = d3.scale.linear()
        .domain([min, max])
        .range("#fff", "#0f0");
var points = globe.points()
        .data(function(d)       {
                return datasets[d];
        })
        .latitude(function(e) {
                return e['latitude'];
        })
        .longitude(function(e) {
                return e['longitude'];
        })
        .color(function(e) {
                return c(e['value']);
        })
        .radius(0.1); // degrees
});
```

Figure 10: Creating data points

Internal Implementation

In each frame (ie, 60 times per second), we iterate over all data elements and draw filled circles to a hidden canvas element, using user-defined values for the location, color, and size of the circle. The canvas is bound to a texture which is passed into the fragment shader as a uniform Sampler2D. The fragment shader then has two textures it is sampling from – the base texture for the globe, and the data texture with the points. The points are overlaid on top of the background texture using the GLSL `mix` function to achieve the desired effect.

Handling mouse events on points involves figuring out which data element corresponds to the latitude and longitude of the mouse position, which is obtained using the methods described in the previous section. For this, the event handler iterates over all data elements and figures out the closest element that is within a threshold difference in angle. As discussed previously, this data element is then stored as a property of the mouse event.

Painter

Painter lets the client specify a function that is passed the 2D context of the hidden canvas discussed in the previous section. This gives the client the freedom to draw any type of image, path, shapes, or text that will be overlaid on the globe. To create a painter, the client binds data and provides a canvas painting function:

```
var painter = globe.painter()
        .data(function(d) { return datasets[d]; })
        .paint(function(gl, context, datum) {
                // draw on context
```

```
});
```

Figure 11: Create a painter with a client-defined painting function

Figure 12: Painter is used to draw paths on globe In D3GL demo, in which points is defined on top of painter to provide interaction

Shapes

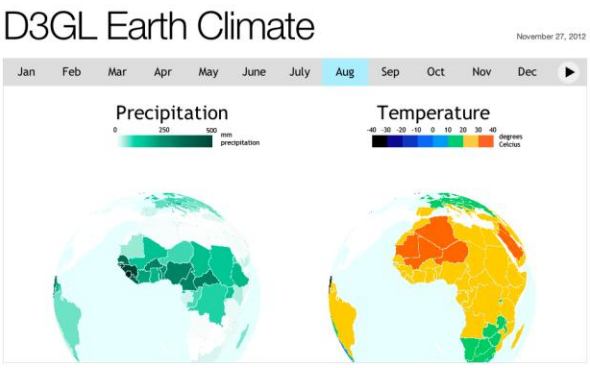


Figure 13: Shapes encode world climate data in D3GL demo

The shapes primitive colors user-defined shapes with different colors according to the value of the corresponding data element. The initial purpose for `shapes` was to create choropleth maps of the world, but later we defined "countries" as type of shapes primitive the user could choose, opening up possibilities for visualization of and interaction with user-defined shapes.

The following example creates a `shapes` overlay for different countries, colors the countries varying shades of gray according to the values of the corresponding data elements, and highlights the selected country yellow:

```
var selected;
var c = d3.scale.linear()
        .domain([min, max])
        .range(0, 255);
var shapes = globe.shapes("countries")
        .data(function(d) { return datasets[d]; })
        .id(function(e) { return e['id'] })
        .color(function(e) {
                if(selected==e) {
                        return [255, 255, 0, 255];
                }
                var color = c(e['value']);
                return [color, color, color, 255];
        })
```

Figure 14: Creating data shapes

The call to `data` works as described in the previous sections. The function `id` returns a unique id for each data element, which also corresponds to a shape. In Figure 14, since the default shapes "countries" is specified, each data element that represents a country should have the id function return the country's ISO numeric code. The color function must return an array [R,G,B,A], where each component is a byte. In the future, D3GL may support HTML color return values like "#ff0000".

For custom shapes, the user can pass in the URL of an image that specifies different shapes rather than "countries":

```
var shapes = globe.shapes("shapes.png")
```

Figure 15: Specifying custom shapes

In order to make this work, `shapes.png` in Figure 15 needs to color-code up to 1000 different shapes, with corresponding ids between 0 and 999. The color needs to represent the id of the data element that corresponds to the shape such that the color for id `ijk` (base 10) is `rgb(i*25.5, j*25.5, k*25.5)`. For example, in the color-coded map that is used when the user specifies "countries", the United States, which has the ISO numeric code of 840, is colored `rgb(8*25.5, 4*25.5, 0*25.5)`. If the user decides to pass in a custom color-coded map of all the countries in the world using a different id system, the colors would be different and the id function would differ correspondingly.

Internal Implementation

Under the hood, `shapes` loads the given shapes image as texture and passes it in to the fragment shader as a uniform Sampler2D. For each frame, `shapes` iterates over each data element, gets the user-specified color for the element, and stores this information in a data texture that maps ids to color. This data texture is

passed into the fragment shader as another uniform Sampler2D. In the shader, the appropriate color for the shape that the fragment belongs to is computed by looking up the shapes texture to find the color that represents the id of the shape, converting the color into an integer id, and using the id to look up into the data texture to get the color for the shape. Then this color is mixed with the appropriate background texture and overlay texture color to produce the final shade for the fragment.

Selection of shapes works like that in points, the variable shapes in the following snippet of code being the one initialized in Figure 10:

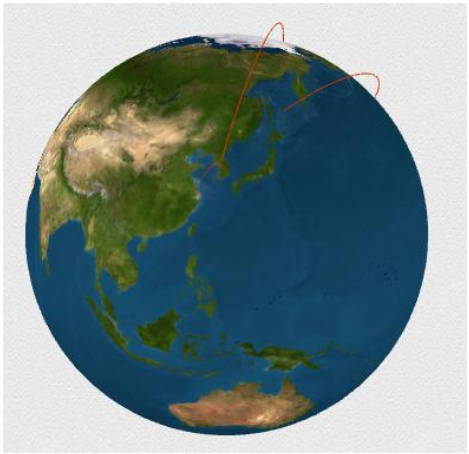
```
var clicked;
shapes.on("click", function(evt) {
        clicked = evt.shape;
});
```

Figure 16: Handling mouse events for shapes

In Figure 16, every time the user clicks on the globe, the variable `clicked` is assigned the data element that is clicked on, or null if the user clicked on a region that did not belong to any shape.

This is implemented by raytracing the mouse coordinates as described above to get (lat,lon), then mapping this to the corresponding pixel coordinates for the shapes texture. Then, the texel is read, and its color translated into the id for a data element. Finally, `shapes` iterates through all data elements and assigns to `evt.shape` the element whose id matches the computed one, or null if none is found. This runs fast enough to support smooth mouseover/hover interactions on large shape textures.

Arcs



The arcs display is useful for visualizing connections or flows between points on the globe. For example, it could be used to encode airline routes or international trade statistics.

```
var h = d3.scale.linear()
        .domain([min, max])
        .range([0, 0.5]);
var bars = globe.arcs()
        .data(flights)
        .id(function(d) { return d['id']; })
        .color(function(d){return d.airlineColor})
        .apex(0.05) // 0.05 globe radii
        .start(function(d) { return [
            d['departure'].lat,
            d['departure'].lon
         ]})
        .end(function(d) { return [
            d['arrival'].lat,
            d['arrival'].lon
         ]});
```

Figure 17: Creating arcs

The arcs overlay allows developers to map data to a start and end point on the globe. The optional apex() property controls how high the midpoint of the arc is above the globe surface, measured in globe radii. The optional partialArc() property must return a value between 0 and 1, and values less than 1 show an arc that ends before its destination. This property is useful, for example, for visualizing flights in progress.

The arcs primitive is especially effective compared to 2D globe projections, which have a high "lie factor" when representing distances. For example, a flight from Los Angeles to London looks strange on a Mercator projection—it is dramatically curved, travelling over Iceland. On a globe, it is much clearer that this is, in fact, the most direct route.

Internal Implementation

Unlike `points` and `shapes`, which create overlay textures that are passed into the fragment shader to determine the final color for the fragment, `arcs` creates new 3D elements.

It renders WebGL line strip primitives with a flat-color shader. The arcs primitive also computes the great-circle (global shortest distance) path between each pair of endpoints.

Bars

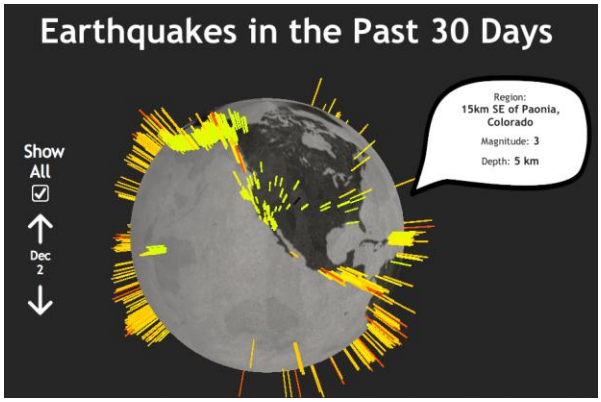


Figure 18: Bars encode global earthquake data in D3GL demo

Bars on a 3D globe are useful for showing location data with a magnitude for each location. Three-dimensional bar charts are often cited as examples of 3D data visualization gone wrong. This refers to Excel-style 3D charts, where the third dimension is redundant. When you have a third or fourth data dimension, however, a spatial visualization can help. This visualization is interactive, alleviating occlusion problems. Bars allow better comparison of quantitative variables than, say, a color coding.

Creating a bar overlay follows the same convention as the other primitives:

```
var h = d3.scale.linear()
        .domain([min, max])
        .range([0, 0.5]);
var bars = globe.bars()
        .data(function(d) { return datasets[d]; })
        .id(function(d) { return d['id']; })
        .width(0.01)
        .height(function(e) {
                return h(e['value']);
        })
        .color("#ff0000");
```

Figure 19: Creating bars

The width and height are in units of the globe's radius.

Internal Implementation

Each bar is a Three.js `CubeGeometry`, where the vertices and rotation are set according to the bar's location, width, and height. Thus, `bars` cannot afford to create the elements anew per frame as happens with points and shapes – it would be an unnecessary burden on memory. Instead, `bars` requires that each data element be given a unique id as determined by the `id` call in Figure 19. This id is used to look up whether a bar geometry has been created that represents the data element. Only when the data element does not have a corresponding 3D geometry is a new bar created. Otherwise, the already-created bar is found and its dimensions and color updated.

The only exception is when a new dataset is passed in, replacing the old dataset. All the old bars are removed, and thus new datasets can be introduced that have data elements whose ids may overlap those of elements in the old dataset.

Thus far, `bars` is the only D3GL data primitive to support transition. In `points` or `shapes`, any transition would have to be created by the user, including management of frames and interpolated values. With bars, it is possible to specify transition between heights as in the following, which assumes Figure 19:

```
bars.transition()
                .delay(60)
                .duration(60)
                .ease("sin")
                .height(0)
        .transition()
                .delay(60)
                .duration(30)
                .height(function(e) {
                        return h(e['value']);
                });
```

Figure 20: Transition with bar heights

When the above code is executed, there will be a 60-frame delay before all bars transition to a height of 0 in another 60 frames according to the d3 sin wave ease function as specified. Then there will be another 60-frame delay before they take 30 frames to return to their original height, this time linearly, as happens when no ease function is specified.

Each call to `transition()` pushes a transition object to the back of a queue. Any setter invoked after the `transition` call will add the value or function to the transition object at the end of the queue. In each frame, an update call determines the next transition to run. When a transition first stars, a `t` value is initialized to 0. It grows from 0 to 1 where 0 is the beginning of the transition and 1 the end. If there is a delay value greater than 1, the delay is decreased and the `t` value does not grow. Otherwise, the `t` value is increased by the appropriate amount per frame. Then for each bar, the `t` value of updated transition object is used to compute its current height. When `t` reaches 1, the transition object is removed from the queue.

FUTURE WORK

The globe and the different types of data primitives that can be overlaid on the globe are only the beginning of what we want D3GL to do. We have a couple additional features in mind:

`d3.gl.mesh`

D3GL will support data visualization on arbitrary 3D meshes, such as for CAD models. Example use cases for `d3.gl.mesh` include visualizing stress test data on 3D products and visualizing design iterations using small multiples. Overlays on the 3D meshes will include interactive points, shapes, and heat maps.

`d3.gl.points`

D3GL will also support point data, including point cloud data and multi-dimensional scatter plots.

There are existing applications that let users upload and visualize their own point cloud data, such as http://webgl.uni-hd.de/pointCloudViewer. As far as we know, however, none support interaction with the data other than zoom and rotation. We wish to implement interactions such as selection, brushing, and filtering.

Three-dimensional scatter plots can be useful for exploring multi-dimensional data. D3GL will support a simple and interactive 3D scatter plot with the ability to toggle between different orthographic projections smoothly so as to guide the user between different "flattened" views of the data.

CONCLUSION

D3GL is a D3 extension for spatial data. While D3 provides a powerful and intuitive way to bind data to DOM elements (such as HTML or SVG), our goal is to give developers a way to bind data to 3D primitives.

We started by creating primitives for global, location-based data. For example: data about Earth, its climate, its countries and cities, data about other planets, data about links or trade flows between countries, and so on. We consider the D3GL globe tools to be feature-complete.

The D3GL globe supports point, shape, bar, and arc overlays, as well as a flexible Canvas2D painter overlay. Features include zooming and rotation for navigating, transparency, mouse events including 3D picking, and animation.

The goal of D3GL is to make effective 3D data visualization easy. Documentation and demos are available at http://d3gl.org. All code is open-source and available on Github.

REFERENCES

WebGL Globe by Chrome
http://www.chromeexperiments.com/globe
DataAppeal
http://dataappeal.com/
PhiloGL
http://www.senchalabs.org/philogl/
Michael Bostock. "Towards Reusable Charts"
http://bost.ocks.org/mike/chart/