Using Layers

The "Layer" is a core concept of deck.gl. A deck.gl layer is a packaged visualization type that takes a collection of datums, associate each with positions, colors, extrusions, etc., and renders them on a map.

deck.gl provides an extensive layer catalog and is designed to compose many layers together to form complex visuals.

Constructing a Layer Instance

A layer is instantiated with a `properties` object:

```
import {ScatterplotLayer} from 'deck.gl';

const layer = new ScatterplotLayer({
  id: 'bart-stations',
  data: [
    {name: 'Colma', passengers: 4214, coordinates: [-122.466233, 37.684638]},
    {name: 'Civic Center', passengers: 24798, coordinates:
[-122.413756,37.779528]},
    ...
  ],
  stroked: false,
  filled: true,
  getPosition: d => d.coordinates,
  getRadius: d => Math.sqrt(d.passengers),
  getFillColor: [255, 200, 0]
});
```

The `properties` are settings that the layer uses to build the visualization. Users of a layer typically specify the following types of props:

Layer ID

The `id` prop is the unique identifier of this layer among all layers. Constructing a new layer instance in its own does not have any performance impact, as deck.gl only does the expensive calculations when a layer is **created** (an id appearing for the first time) or **updated** (different props are passed in for the same id). Read more about this in layer lifecycle.

It is recommend that this prop is set explicitly to avoid collision.

Data

The `data` prop specifies data source of this layer's visualization. The value is expected to be a collection (typically a JavaScript array) of data objects with similar structure, such as rows in a table. deck.gl layers are able to handle millions of data objects very efficiently.

The value of this prop can be `Array`, `Map`, `Set`, any object that contains a `length` field, a `Promise` that resolves to any of the above, or an URL to a JSON array. See data prop documentation for details.

Accessors

An accessor is a prop that maps an object in `data` to its visual configuration, e.g. the radius of a circle, the color of a line, etc. All accessor prop names start with `get`.

If an accessor prop is set to a function, when the layer is about to be drawn on screen for the first time, the layer will traverse the `data` stream and call the accessor function with each element.

The accessor function receives two arguments:

- `object` - the current element in the data stream. If `data` is an array or an iterable, the element of the current iteration is used. If `data` is a non-iterable object, this argument is always `null`.
- `objectInfo` (Object) - contextual information of the current element. Contains the following fields:
  - `index` (Number) - the index of the current iteration
  - `data` - the value of the `data` prop
  - `target` (Array) - a pre-allocated array. The accessor function can optionally fill data into this array and return it, instead of creating a new array for every object. In some browsers this improves performance significantly by reducing garbage collection.

The accessor function is typically expected to return either a number or an array.

Some accessors also support constant values instead of functions. When a constant value is provided, it is applied to all objects in the data stream. Unlike a functional value, which is called once for every data object to fill a WebGL buffer proportional to the data size, constant accessors have CPU and memory cost of `O(1)` and are much more performant.

Other Layer Props

The rest of the props are typically numeric or boolean values that apply to the whole layer. These include props that define the render options (opacity, extrusion of the PolygonLayer, font family of the TextLayer, etc.), coordinate system, and interactivity.

Rendering Layers

deck.gl allows you to render multiple layers using the same or different data sets. You simply provide an array of layer instances and deck.gl will render them in order (and handle interactivity when hovering clicking etc).

This allows you to compose visualizations using several primitive layers.

```
import {PathLayer, ScatterplotLayer, ArcLayer} from '@deck.gl/layers';

// A layers array contains deck.gl layer instances. It can also be nested, or
contain empty objects
const layers = [
  new PathLayer({id: 'paths', data: ...}),
  shouldRenderPoints ? [
    new ScatterplotLayer({id: 'big-points', data: ...}),
    new ScatterplotLayer({id: 'small-points', data: ...})
  ] : null,
  new ArcLayer({id: 'arcs', data: ...})
];
```

```
// Use with pure JS
import {Deck} from '@deck.gl/core';
const deck = new Deck({...});
deck.setProps({layers});

// Use with React
import DeckGL from '@deck.gl/react';
<DeckGL ... layers={layers} />
```

FAQ

Should I be Creating New Layers on Every Render?

The Reactive Programming Paradigm

deck.gl's architecture is based on the reactive programming paradigm:

- In a reactive application, a complete UI description is "re-rendered" every time something in the application state changes (in the case of a deck.gl application, a new list of layers is created whenever something changes).
- The UI framework (in this case, deck.gl) makes the choices about what to update, by comparing (or "diffing") the newly rendered UI description with the last rendered UI description.
- The framework then the makes minimal necessary changes to account for the differences, and then redraws.
- The required changes are made to "WebGL state" in case of deck.gl, and to the Browser's DOM (HTML element tree) in case of React.

Creating Layer Instances Is Cheap

The deck.gl model means that applications are expected to create a new set of layers every time application state changes, which can seem surprisingly inefficient to someone who hasn't done reactive programming before. The trick is that layers are just descriptor objects that are very cheap to instantiate, and internally, the new layers are efficiently matched against existing layers so that no updates are performed unless actually needed.

So, even though the application creates new "layers", those layers are only "descriptors" containing props that specify what needs to be rendered and how. All calculated state (WebGL "programs", "vertex attributes" etc) are stored in a state object and this state object is moved forward to the newly matched layer on every render cycle. The new layer ends up with the state of the old layer (and the props of the new layer), while the old layer is simply discarded for garbage collection.

The application does not have to be aware about this, as long as it keeps rendering new layers with the same `id` they will be matched and the existing state of that layer will be updated accordingly.

The constant creation and disposal of layer instances may seem wasteful, however the creation and recycling of JavaScript objects is quite efficient in modern JavaScript environments, and this is very similar to how React works where every render cycle generates a new tree of ReactElement instances, so the model is proven.

For more details on layer creation, update and destruction, read about Layer Lifecycle.

Why Doesn't My Layer Update with New Props?

Since the reactive programming frameworks conceptually render the entire UI every render cycle, and achieves efficiency by comparing and "diffing" changes between render cycles, it is important that comparisons are fast. Because of this, deck.gl uses shallow equality as the default comparison method for most props.

An exception is the accessors. Changing the value of an accessor (i.e. supplying a different function to the accessor prop) will not in itself trigger an attribute update. This is because the function identity is a poor indicator of whether an update is needed, and the convenience of using local functions as prop values.

Consider the code below, `getRadius` is shallowly changed every time `render()` is called, even if its execution result would not change:

```
function render() {
  const layer = new ScatterplotLayer({
    ...
    getColor: x => x.color, // this creates a new function every render
    getRadius: this._getRadius.bind(this) // bind generates a new function every
render
  });
  deck.setProps([layer]);
}
```

In another example, `getColor` is shallowly changed, so would its execution result:

```
const DATA = [...];

function setPillColor(index, color) {
  DATA[index].pill = color;
  render();
}

function render() {
  const layer = new ScatterplotLayer({
    data: DATA,
    ...
    getColor: object => object.pill === 'red' ? [255, 0, 0] : [0, 0, 255], //
Does not trigger an attribute update!
  });
  deck.setProps([layer]);
}
```

There is no way for deck.gl to know what the programmer intended just by looking at or comparing the functions that are supplied to a `Layer`s accessor props. Because recalculating attributes can be potentially expensive, deck.gl by default ignores shallow changes in accessor props. This is designed to provide the best performance to the most common use cases without compromising convenience.

Instead, the `updateTriggers` property gives you fine grained control, enabling you to tell deck.gl exactly which attributes need to change, and when.

Read more about this behavior in Performance Optimization.

Edit this page