Adding Interactivity

Controlling the Camera

Out of the box, deck.gl offers viewport controllers that map keyboard, mouse or touch input to camera state change. The easiest way to enable pan/zoom/rotate of the visualization is to set the `controller` prop on `Deck` or `<DeckGL>` to `true` along with an `initialViewState` object that defines the initial camera settings:

```
import {Deck} from '@deck.gl/core';

new Deck({
  initialViewState: {
    longitude: -122.4,
    latitude: 37.8,
    zoom: 12,
    pitch: 0,
    bearing: 0
  },
  controller: true,
  layers: []
});
```

You can also selectively enable/disable certain controller features:

```
  controller: {doubleClickZoom: false, touchRotate: true}
```

See Controller for all options.

Reset Camera Position

An application can reset the camera state by supplying a new `initialViewState` object at any time:

```
import {Deck} from '@deck.gl/core';
```

```
const deckgl = new Deck({
  initialViewState: {
    longitude: -122.4,
    latitude: 37.8,
    zoom: 12
  },
  controller: true,
  layers: []
});

button.onclick = gotoNYC;

// Jump to New York City
function goToNYC() {
  deckgl.setProps({
    initialViewState: {
      longitude: -70.4,
      latitude: 40.7,
      zoom: 12
    }
  })
}
```

To add a transition animation, see view state transitions.

Add Constraints to View State

An application can optionally supply the onViewStateChange callback and manipulate the view state before it is used. The following example constrains the map in a bounding box:

```
import {Deck} from '@deck.gl/core';

const LONGITUDE_RANGE = [-123, -122];
const LATITUDE_RANGE = [37, 38];

new Deck({
  initialViewState: {
    longitude: -122.4,
    latitude: 37.8,
    zoom: 12
  },
  controller: true,
  onViewStateChange: ({viewState}) => {
    viewState.longitude = Math.min(LONGITUDE_RANGE[1],
Math.max(LONGITUDE_RANGE[0], viewState.longitude));
```

```
    viewState.latitude = Math.min(LATITUDE_RANGE[1], Math.max(LATITUDE_RANGE[0],
viewState.latitude));
    return viewState;
  }
});
```

Externally Manage View State

For more flexibility you can maintain the view state yourself and pass it in to deck.gl via the `viewState` parameter. This essentially makes `Deck`/`<DeckGL>` a stateless component, and allows you to share the view state between multiple components, e.g. via a Redux store.

Note: Do not combine `initialViewState` and `viewState` props. `viewState` will always overwrite any internal state.

The following example demonstrates how to do this with React:

```
import React, {useState} from 'react';
import DeckGL from '@deck.gl/react';

function App() {
  const [viewState, setViewState] = useState({
    longitude: -122.4,
    latitude: 37.8,
    zoom: 12
  });

  const layers = [
    //...
  ];
  return (
    <DeckGL
        viewState={viewState}
        onViewStateChange={e => setViewState(e.viewState)}
        controller={true}
        layers={layers}>
    </DeckGL>
  );
}
```

Advanced View Controls

- Alternative views such as OrbitView, FirstPersonView, and using multiple views such as VR,

minimap: Views and Projections
- Implement a custom controller: Controller

Picking

deck.gl includes a powerful picking engine that enables the application to precisely determine what object and layer is rendered on a certain pixel on the screen. This picking engine can either be called directly by an application (which is then typically implementing its own event handling), or it can be called automatically by the basic built-in event handling in deck.gl

What can be Picked?

The "picking engine" identifies which object in which layer is at the given coordinates. While usually intuitive, what constitutes a pickable "object" is defined by each layer. Typically, it corresponds to one of the data entries that is passed in via `prop.data`. For example, in Scatterplot Layer, an object is an element in the `props.data` array that is used to render one circle. In GeoJson Layer, an object is a GeoJSON feature in the `props.data` feature collection that is used to render one point, path or polygon.

Enabling Picking

Picking can be enabled or disabled on a layer-by-layer basis. To enable picking on a layer, set its `pickable` prop to `true`. This value is `false` by default.

The Picking Info Object

The picking engine returns "picking info" objects which contains a variety of fields describing what layer and object was picked.

Key	Value
`layer`	The layer that the picked object belongs to. Only layers with the `pickable` prop set to true can be picked.
`index`	The index of the object in the layer that was picked.

Key	Value
`object`	The object that was picked. This is typically an entry in the layer's `props.data` array, but can vary from layer to layer.
`x`	Mouse position x relative to the viewport.
`y`	Mouse position y relative to the viewport.
`coordinate`	Mouse position in geospatial coordinates. Only applies if `layer.props.coordinateSystem` is a geospatial mode such as `COORDINATE_SYSTEM.LNGLAT`.
`viewport`	The viewport that the picked object belongs to.

Remarks:

- Specific deck.gl Layers may add additional fields to the picking `info` object. Check the documentation of each layer.
- Limitation when using multiple views: `viewport` could potentially be misidentified if two views that contain the picked layer also overlap with each other and do not clear the background.

Example: Display a Tooltip for Hovered Object

Using the Built-In Tooltip

`Deck` automatically renders a tooltip if the `getTooltip` callback is supplied:

```
import {Deck} from '@deck.gl/core';
import {ScatterplotLayer} from '@deck.gl/layers';

const deck = new Deck({
  canvas: 'deck-canvas',
  initialViewState: {longitude: -122.45, latitude: 37.78, zoom: 12},
  controller: true,
  layers: [
    new ScatterplotLayer({
      data: [
        {position: [-122.45, 37.78], message: 'Hover over me'}
      ],
      getPosition: d => d.position,
      getRadius: 1000,
      getFillColor: [255, 255, 0],
      // Enable picking
      pickable: true
    })
  ],
  getTooltip: ({object}) => object && object.message
});
```

It receives a picking info object and returns the content of the tooltip. To custom the tooltip further, return an object instead:

```
  getTooltip: ({object}) => object && {
    html: `<h2>${object.name}</h2><div>${object.message}</div>`,
    style: {
      backgroundColor: '#f00',
      fontSize: '0.8em'
    }
  }
```

For a range of options, see getTooltip documentation.

Using React

```
import React, {useState} from 'react';
import {DeckGL, ScatterplotLayer} from 'deck.gl';

const data = [
  {position: [-122.45, 37.78], message: 'Hover over me'}
];

function App() {
  const [hoverInfo, setHoverInfo] = useState;

  const layers = [
    new ScatterplotLayer({
      data,
      getPosition: d => d.position,
      getRadius: 1000,
      getFillColor: [255, 255, 0],
      // Enable picking
      pickable: true,
      // Update app state
      onHover: info => setHoverInfo(info)
    })
  ];

  return (
    <DeckGL initialViewState={{longitude: -122.45, latitude: 27.78, zoom: 12}}
        controller={true}
        layers={layers} >
      {hoverInfo.object && (
        <div style={{position: 'absolute', zIndex: 1, pointerEvents: 'none',
left: hoverInfo.x, top: hoverInfo.y}}>
          { hoverInfo.object.message }
        </div>
      )}
    </DeckGL>
  );
}
```

Calling the Picking Engine Directly

The picking engine is exposed through the `Deck.pickObject` and `Deck.pickObjects` methods. These methods allow you to query what layers and objects within those layers are under a specific point or within a specified rectangle. They return `Picking Info` objects as described below.

`pickObject` allows an application to define its own event handling. When it comes to how to actually do event handling in a browser, there are many options. In a React application, perhaps the simplest is to just use React's "synthetic" event handling together with `pickObject`:

```
import React, {useRef, useCallback} from 'react';

function App() {
  const deckRef = useRef(null);

  const onClick = useCallback(event => {
    const pickInfo = deckRef.current.pickObject({
      x: event.clientX,
      y: event.clientY,
      radius: 1
    });
    console.log(pickInfo.coordinate);
  }, [])
```

```
    return (
      <div onClick={onClick}>
        <DeckGL ref={deckRef} ... />
      </div>
    );
  }
```

Also note that by directly calling `queryObject`, integrating deck.gl into an existing application often becomes easier since you don't have to change the application's existing approach to event handling.

Under The Hood

If you are using the core layers, all has been taken care of.

If you are implementing a custom layer, read more about how picking is implemented.

Built-in Events

For applications that have basic event handling needs, deck.gl has built-in support for handling selected pointer events. When the application registers callbacks, deck.gl automatically tracks these events, runs the picking engine and calls application callbacks with a single parameter `info` which contains the resulting picking info object.

The following event handlers are supported:

- `onHover`
- `onClick`
- `onDragStart`
- `onDrag`
- `onDragEnd`

A event handler function is called with two parameters: `info` that contains the object being interacted with, and `event` that contains the pointer event.

There are two ways to subscribe to the built-in picking event handling:

- Specify callbacks for each pickable layer by passing event handler props:

```
const layer = new ScatterplotLayer({
    ...
    pickable: true,
```

```
<DeckGL
    ...
    onHover={this._onHover}
    onClick={this._onClick}
/>
```

Behavior of Built-in Event Handling

Picking events are triggered based on *pickable objects*:

- A `click` event is triggered every time the pointer clicked on an object in a pickable layer.
- A `hover` event is triggered every time the hovered object of a pickable layer changes.

When an event is fired, the `onHover` or `onClick` callback of the affected layer is called first. If the callback returns a truthy value, the event is marked as handled. Otherwise, the event will bubble up to the `DeckGL` canvas and be visible to its `onHover` and `onClick` callbacks.

Edit this page