Animations and Transitions

> This document was recently added and is still being finalized

deck.gl provides several ways to implement animation and transitions.

- **Camera Transitions** - (aka View State Transitions) when changing the view state, deck.gl can move the camera smoothly between the initial and final view state.
- **Layer Transitions** - (aka Attribute Transitions)
- **Property Animation** -

Animation

Drawing Constantly vs. When Needed

While some 3D applications such as games often keep drawing to screen at a high frame rate, deck.gl avoids this by default. When not animating, deck.gl is optimized to only render to the screen when something changes (a layer's data or props, the viewState etc). This keeps the GPU and CPU load low and minimizes power consumption.

However when animating properties, deck.gl needs to update the screen frequently. This does consume more power and can cause fans to spin up etc. However, even when animating deck.gl does not draw to the screen when the application's browser tab is not active.

Property Animation

Layer properties are divided into two categories. Accessors (that control "attributes" and only update when data changes or update triggers fire) and non-accessors that update every time the layer is rendered.

Property animation refers to assigning functions as values to non-accessor layer props. The functions will be called every render frame, potentially generating new values every frame at 60 FPS.

Advantages of Property Animation

While it is certainly possible to implement deck.gl animation by supplying a "freshly minted" list of layers to the deck.gl component every frame, e.g. using `Deck.setProps({layers: [new ...Layer(...), ...]})`, this approach has a performance penalty as both your application logic and deck.gl's layer matching system must kick into gear each render frame.

By setting property animation functions, properties will be evaluated by the underlying render loop system with considerably less overhead, both deck.gl and your application will essentially remain undisturbed as the layer properties are updated inside a tight inner animation loop.

Transitions

Transitions in deck.gl are short animations that are triggered when deck.gl detects a change of some property. Instead of immediately redrawing based on the new view value, deck.gl draws a number of frames, automatically interpolating the value over time.

View State Transitions

View state transitions provide smooth and visually appealing camera transitions when view states change. View state transitions are initiated by updating the `Deck.viewState` prop.

Following fields of `viewState` can be used to achieve viewport transitions.

- `transitionDuration` (Number, optional, default: 0) - Transition duration in milliseconds, default value 0, implies no transition.
- `transitionEasing` (Function, optional, default: `t => t`) - Easing function that can be used to achieve effects like "Ease-In-Cubic", "Ease-Out-Cubic", etc. Default value performs Linear easing. (list of sample easing functions: http://easings.net/)
- `transitionInterpolator` (Object, optional, default: `LinearInterpolator`) - An interpolator object that defines the transition behavior between two viewports, deck.gl provides `LinearInterpolator` and `FlyToInterpolator`. Default value, `LinearInterpolator`, performs linear interpolation on `ViewState` fields. `FlyToInterpolator` animates `ViewStates` similar to MapBox `flyTo` API and applicable for `MapState`, this is pretty useful when camera center changes by long distance. But a user can provide any custom implementation for this object using `TrasitionInterpolator` base class.
- `transitionInterruption` (TRANSITION_EVENTS (Number), optional, default: BREAK) - This field controls how to process a new `ViewState` change that occurs while performing

an existing transition. This field has no impact once transition is complete. Here is the list of all possible values with resulting behavior.

TRANSITION_EVENTS	Result
BREAK	Current transition will stop at the current state and next `ViewState` update is processed.
SNAP_TO_END	Current transition will skip remaining transition steps and `ViewState` is updated to final value, transition is stopped and next `ViewState` update is processed.
IGNORE	Any `ViewState` update is ignored until current transition is complete, this also includes `ViewState` changes due to user interaction.

- `onTransitionStart` (Functional, optional) - Callback fires when requested transition starts.
- `onTransitionInterrupt` (Functional, optional) - Callback fires when transition is interrupted.
- `onTransitionEnd` (Functional, optional) - Callback fires when transition ends.

TransitionInterpolators

- **LinearInterpolator** - Performs linear interpolation between two ViewStates.
- **FlyToInterpolator** - This class is designed to perform `flyTo` style interpolation between two `MapState` objects.
- **TransitionInterpolator** - Base interpolator class that provides common functionality required to interpolate between two ViewState props. This class can be subclassed to implement any custom interpolation:

Examples

Sample code that provides `flyTo` style transition to move camera from current location to NewYork city.

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      viewState: {
        latitude: 37.7751,
        longitude: -122.4193,
        zoom: 11,
        bearing: 0,
        pitch: 0,
        width: 500,
        height: 500
      }
    };
    this._onViewStateChange = this._onViewStateChange.bind(this);
  }

  _goToNYC() {
    this.setState({
      viewState: {
        ...this.state.viewState,
        longitude: -74.1,
        latitude: 40.7,
        zoom: 14,
        pitch: 0,
        bearing: 0,
        transitionDuration: 8000,
        transitionInterpolator: new FlyToInterpolator()
      }
    });
  }

  _onViewStateChange({viewState}) {
    this.setState({viewState});
  }

  render() {
    const {viewState} = this.state;

    return (
      <div>
        <DeckGL
          viewState={viewState}
          controller={MapController}
          onViewStateChange={this._onViewStateChange}
        >
          <StaticMap
            // props
            ...
          />
        </DeckGL>

        <button onClick={this._goToNYC}>New York City</button>
      </div>
    );
  }
}
```

Sample code to get continuous rotations along vertical axis until user interrupts by rotating the map by mouse interaction. It uses `LinearInterpolator` and restricts transitions for `bearing` prop. Continuous transitions are achieved by triggering new transitions using `onTranstionEnd` callback.

```
const transitionInterpolator = new LinearInterpolator(['bearing']);

const INITIAL_VIEW_STATE = {
  // set to required initial view state
...
};

class App extends Component {
  constructor(props) {
    super(props);
    this.rotationStep = 0;
    this.state = {
      viewState: INITIAL_VIEW_STATE
    };

    this._onLoad = this._onLoad.bind(this);
    this._onViewStateChange = this._onViewStateChange.bind(this);
    this._rotateCamera = this._rotateCamera.bind(this);
  }

  _onLoad() {
    this._rotateCamera();
  }

  _onViewStateChange({viewState}) {
    this.setState({viewState});
  }

  _rotateCamera() {
    // change bearing by 120 degrees.
    const bearing = this.state.viewState.bearing + 120;
    this.setState({
      viewState: {
        ...this.state.viewState,
        bearing,
        transitionDuration: 1000,
        transitionInterpolator,
        onTransitionEnd: this._rotateCamera
      }
    });
  }

  _renderLayers() {
    // render any deck.gl layers
    ...
  }

  render() {
    const {viewState} = this.state;
    return (
      <DeckGL
        layers={this._renderLayers()}
        viewState={viewState}
        onLoad={this._onLoad}
        onViewStateChange={this._onViewStateChange}
        controller={true}
      >
        <StaticMap
          // props
          ...
        />
      </DeckGL>
    );
  }
}
```

Edit this page