



PathLayer

[< > Edit on Codepen](#)

The `PathLayer` renders lists of coordinate points as extruded polylines with mitering.

```
import DeckGL from '@deck.gl/react';
import {PathLayer} from '@deck.gl/layers';

function App({data, viewState}) {
  /**
   * Data format:
   * [
   *   {
   *     path: [[-122.4, 37.7], [-122.5, 37.8], [-122.6, 37.85]],
   *     name: 'Richmond - Millbrae',
   *     color: [255, 0, 0]
   *   },
   *   ...
   */
}
```

```
* ]
*/
const layer = new PathLayer({
  id: 'path-layer',
  data,
  pickable: true,
  widthScale: 20,
  widthMinPixels: 2,
  getPath: d => d.path,
  getColor: d => colorToRGBArray(d.color),
  getWidth: d => 5
});

return <DeckGL viewState={viewState}
  layers={[layer]}
  getTooltip={({object}) => object && object.name} />;
}
```

Installation

To install the dependencies from NPM:

```
npm install deck.gl
# or
npm install @deck.gl/core @deck.gl/layers
```

```
import {PathLayer} from '@deck.gl/layers';
new PathLayer({});
```

To use pre-bundled scripts:

```
<script src="https://unpkg.com/deck.gl@8.0.0/dist.min.js"></script>
<!-- or -->
<script src="https://unpkg.com/@deck.gl/core@8.0.0/dist.min.js"></script>
<script src="https://unpkg.com/@deck.gl/layers@8.0.0/dist.min.js"></script>
```

```
new deck.PathLayer({});
```

Properties

Inherits from all [Base Layer](#) properties.

Render Options

`widthUnits` (**String, optional**)

- Default: `'meters'`

The units of the line width, one of `'meters'`, `'common'`, and `'pixels'`. See [unit system](#).

`widthScale` (**Number, optional**) transition enabled

- Default: `1`

The path width multiplier that multiplied to all paths.

`widthMinPixels` (**Number, optional**) transition enabled

- Default: `0`

The minimum path width in pixels. This prop can be used to prevent the path from getting too thin when zoomed out.

`widthMaxPixels` (**Number, optional**) transition enabled

- Default: `Number.MAX_SAFE_INTEGER`

The maximum path width in pixels. This prop can be used to prevent the path from getting too thick when zoomed in.

`capRounded` (**Boolean, optional**)

- Default: `false`

Type of caps. If `true`, draw round caps. Otherwise draw square caps.

`jointRounded` (**Boolean, optional**)

- Default: `false`

Type of joint. If `true`, draw round joints. Otherwise draw miter joints.

`billboard` (**Boolean, optional**)

- Default: `false`

If `true`, extrude the path in screen space (width always faces the camera). If `false`, the width always faces up.

`miterLimit` (**Number, optional**) transition enabled

- Default: `4`

The maximum extent of a joint in ratio to the stroke width. Only works if `jointRounded` is `false`.

`_pathType` (**Object, optional**)

- Default: `null`

Note: This prop is experimental

One of `null`, `'loop'` or `'open'`.

If `'loop'` or `'open'`, will skip normalizing the coordinates returned by `getPath` and instead assume all paths are to be loops or open paths. Disabling normalization improves performance during data update, but makes the layer prone to error in case the data is malformed. It is only recommended when you use this layer with preprocessed static data or validation on the backend.

When normalization is disabled, paths must be specified in the format of flat array. Open paths must contain at least 2 vertices and closed paths must contain at least 3 vertices. See `getPath` below for details.

Data Accessors

`getPath` (**Function, optional**) transition enabled

- Default: `object => object.path`

Called on each object in the `data` stream to retrieve its corresponding path.

A path can be one of the following formats:

- An array of points (`[x, y, z]`). Compatible with the GeoJSON [LineString](#) specification.
- A flat array or [TypedArray](#) of numbers, in the shape of `[x0, y0, z0, x1, y1, z1, ...]`. By default, each coordinate is assumed to contain 3 consecutive numbers. If each coordinate contains only two numbers (x, y), set the `positionFormat` prop of the layer to `XY`:

```
new PathLayer({
  ...
  getPath: object => object.vertices, // [x0, y0, x1, y1, x2, y2, ...]
  positionFormat: `XY`
})
```

`getColor` ([Function](#)|[Array](#), optional) transition enabled

- Default `[0, 0, 0, 255]`

The rgba color of each object, in `r, g, b, [a]`. Each component is in the 0-255 range.

- If an array is provided, it is used as the color for all objects.
- If a function is provided, it is called on each object to retrieve its color.

`getWidth` ([Function](#)|[Number](#), optional) transition enabled

- Default: `1`

The width of each path, in units specified by `widthUnits` (default meters).

- If a number is provided, it is used as the width for all paths.
- If a function is provided, it is called on each path to retrieve its width.

Use binary attributes

This section is about the special requirements when [supplying attributes directly](#) to a `PathLayer`.

Because each path has a different number of vertices, when `data.attributes.getPath` is supplied, the layer also requires an array `data.startIndices` that describes the vertex index at

the start of each path. For example, if there are 3 paths of 2, 3, and 4 vertices each,

`startIndices` should be `[0, 2, 5, 9]`.

Additionally, all other attributes (`getColor`, `getWidth`, etc.), if supplied, must contain the same layout (number of vertices) as the `getPath` buffer.

To truly realize the performance gain from using binary data, the app likely wants to skip all data processing in this layer. Specify the `_pathType` prop to skip normalization.

Example use case:

```
// USE PLAIN JSON OBJECTS
const PATH_DATA = [
  {
    path: [[-122.4, 37.7], [-122.5, 37.8], [-122.6, 37.85]],
    name: 'Richmond - Millbrae',
    color: [255, 0, 0]
  },
  ...
];

new PathLayer({
  data: PATH_DATA,
  getPath: d => d.path,
  getColor: d => d.color
})
```


Convert to using binary attributes:

```
// USE BINARY
// Flatten the path vertices
// [-122.4, 37.7, -122.5, 37.8, -122.6, 37.85, ...]
const positions = new Float64Array(PATH_DATA.map(d => d.path).flat(2));
// The color attribute must supply one color for each vertex
// [255, 0, 0, 255, 0, 0, 255, 0, 0, ...]
const colors = new Uint8Array(PATH_DATA.map(d => d.path.map(_ =>
d.color)).flat(2));
// The "layout" that tells PathLayer where each path starts
const startIndices = new Uint16Array(PATH_DATA.reduce((acc, d) => {
  const lastIndex = acc[acc.length - 1];
  acc.push(lastIndex + d.path.length);
  return acc;
}, [0]]));
```

```
new PathLayer({
  data: {
    length: PATH_DATA.length,
    startIndices: startIndices, // this is required to render the paths
    correctly!
    attributes: {
      getPath: {value: positions, size: 2},
      getColor: {value: colors, size: 3}
    }
  },
  _pathType: 'open' // this instructs the layer to skip normalization and use
  the binary as-is
})
```

Source

[modules/layers/src/path-layer](#)

 [Edit this page](#)