Deck

`Deck` is a class that takes deck.gl layer instances and viewport parameters, renders those layers as a transparent overlay, and handles events.

If you are using React, you should not use this class directly. Instead you should be rendering the DeckGL React component.

Usage

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

const deck = new Deck({
  initialViewState: {
    longitude: -122.45,
    latitude: 37.78,
    zoom: 12
  },
  controller: true,
  layers: [
    new ScatterplotLayer({data})
  ]
});
```

Constructor

Creates a new Deck instance.

```
const deck = new Deck(props);
```

Parameters:

See the Properties section.

Properties

Initialization Settings

The following properties are used to initialize a `Deck` instance. Any custom value should always be provided to the `Deck` constructor. Changing them with `setProps` afterwards will have no effect.

`canvas` (HTMLCanvasElement | String, optional)

The canvas to render into. Can be either a HTMLCanvasElement or the element id. Will be auto-created if not supplied.

`gl` (WebGLContext)

WebGL context. Will be auto-created if not supplied.

`glOptions` (Object)

Additional options used when creating the WebGLContext. See WebGL context attributes.

`id` (String)

- Default: `'deckgl-overlay'`

ID assigned to the canvas that is created by this `Deck` instance, to allow style customization in CSS.

`parent` (HTMLElement)

- Default: `document.body`

The container to append the auto-created canvas to.

`debug` (Boolean)

- Default: `false`

Flag to enable WebGL debug mode. Also requires an extra luma.gl import:

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

```
new Deck({
  // ...
  debug: true
})
```

Notes:

- Debug mode is slower as it will use synchronous operations to keep track of GPU state.

`_typedArrayManagerProps` (Object)

- Default: {}

(Experimental) May contain the following fields:

- `overAlloc` (Number) - Default `2`. By default, attributes are allocated twice the memory than they actually need (on both CPU and GPU). Must be larger than `1`.
- `poolSize` (Number) - Default `100`. When memory reallocation is needed, old chunks are held on to and recycled. Smaller number uses less CPU memory. Can be any number `>=0`.

The above default settings make data updates faster, at the price of using more memory. If the app does not anticipate frequent data changes, they may be aggressively reduced:

```
new Deck({
  // ...
  _typedArrayManagerProps: isMobile ? {overAlloc: 1, poolSize: 0} : null
})
```

Rendering Configuration

`width` (Number|String)

- Default: `'100%'`

Width of the canvas, a number in pixels or a valid CSS string.

`height` (Number|String)

- Default: `'100%'`

Height of the canvas, a number in pixels or a valid CSS string.

`style` **(Object)**

Additional CSS styles for the canvas.

`useDevicePixels` **(Boolean|Number)**

- Default: `true`

Controls the resolution of drawing buffer used for rendering.

- `true`: `Device (physical) pixels` resolution is used for rendering, this resolution is defined by `window.devicePixelRatio`. On Retina/HD systems this resolution is usually twice as big as `CSS pixels` resolution.
- `false`: `CSS pixels` resolution (equal to the canvas size) is used for rendering.
- `Number` (Experimental): Specified Number is used as a custom ratio (drawing buffer resolution to `CSS pixel` resolution) to determine drawing buffer size, a value less than `one` uses resolution smaller than `CSS pixels`, gives better performance but produces blurry images, a value greater than `one` uses resolution bigger than CSS pixels resolution (canvas size), produces sharp images but at a lower performance.

Note:

- Consider setting to `false` or to a number <=1 if better rendering performance is needed.
- When it is set to a high Number (like, 4 or more), it is possible to hit the system limit for allocating drawing buffer, such cases will log a warning and fallback to system allowed resolution.

`parameters` **(Object)**

Expects an object with WebGL settings. Before each frame is rendered, this object will be passed to luma.gl's `setParameters` function to reset the WebGL context parameters, e.g. to disable depth testing, change blending modes etc. The default parameters set by `Deck` on initialization are the following:

```
{
  blend: true,
  blendFunc: [GL.SRC_ALPHA, GL.ONE_MINUS_SRC_ALPHA, GL.ONE,
GL.ONE_MINUS_SRC_ALPHA],
  polygonOffsetFill: true,
  depthTest: true,
  depthFunc: GL.LEQUAL
}
```

Refer to the luma.gl setParameters API for documentation on supported parameters and values.

```
import GL from '@luma.gl/constants';
new Deck({
  // ...
  parameters: {
    blendFunc: [GL.ONE, GL.ONE, GL.ONE, GL.ONE],
    depthTest: false
  }
});
```

Notes:

- Any WebGL `parameters` prop supplied to individual layers will still override the global `parameters` when that layer is rendered.
- An alternative way to set `parameters` is to instead define the `onWebGLInitialized` callback (it receives the `gl` context as parameter) and call the luma.gl `setParameters` method inside it.

`layers` **(Array)**

- Default: `[]`

The array of Layer instances to be rendered.

Nested arrays are accepted, as well as falsy values (`null`, `false`, `undefined`). This allows applications to do something like:

```
new Deck({
  // ...
  layers: dataSource.map((item, index) => item && [
    new ScatterplotLayer({id: `scatterplot-${index}`, ...}),
    new PolygonLayer({id: `polygon-${index}`, ...})
  ])
});
// layers is in the shape of [[<layer>, <layer>], null, [<layer>, <layer>], ...]
```

`layerFilter` (Function)

- Default: `null`

If supplied, will be called before a layer is drawn to determine whether it should be rendered. This gives the application an opportunity to filter out layers from the layer list during either rendering or picking. Filtering can be done per viewport or per layer or both. This enables techniques like adding helper layers that work as masks during picking but do not show up during rendering.

```
new Deck({
  // ...
  layerFilter: ({layer, viewport}) => {
    if (viewport.id !== 'minimap' && layer.id === 'geofence') {
      // only display geofence in the minimap
      return false;
    }
    return true;
  }
}
```

Receives arguments:

- `layer` (Layer) - the layer to be drawn
- `viewport` (Viewport) - the current viewport
- `isPicking` (Boolean) - whether this is a picking pass
- `cullRect` (Object) - if defined, indicates that only the content rendered to the given rectangle is needed.
- `renderPass` (String) - the name of the current render pass. Some standard passes are:
  - `'screen'` - drawing to screen
  - `'picking:hover'` - drawing to offscreen picking buffer due to pointer move
  - `'picking:query'` - drawing to offscreen picking buffer due to user-initiated query, e.g. calling `deck.pickObject`.
  - `'shadow'` - drawing to shadow map

Returns:

`true` if the layer should be drawn.

Notes:

- `layerFilter` does not override the visibility if the layer is disabled via `visible: false` or `pickable: false` props.
- All the lifecycle methods other than `draw` are still triggered even a if a layer is filtered out using this method.

`views` (Object|Array)

- Default: `new MapView()`

A single `View` instance, or an array of `View` instances.

`View`s represent the "camera(s)" (essentially viewport dimensions and projection matrices) that you look at your data with. deck.gl offers multiple view types for both geospatial and non-geospatial use cases. Read the Views and Projections guide for the concept and examples.

`viewState` (Object)

An object that describes the view state for each view in the `views` prop. For example, the default view's view state is described here:

```
new Deck({
  // ...
  viewState: {
    longitude: -122.45,
    latitude: 37.78,
    zoom: 12,
    pitch: 30,
    bearing: 0
  }
})
```

When using multiple views, the `viewState` is a map from each view id to its respective view state object. See example.

Transitions between two viewState objects can also be achieved by providing set of fields to `viewState` prop, for more details check ViewState Transitions).

Notes:

- If you supply this prop, you are responsible of managing the changes to the view state upon user interaction. This prop is therefore usually used together with the `onViewStateChange` callback (example). For `Deck` to update view states automatically, use the `initialViewState` prop instead.

`initialViewState` (Object)

If `initialViewState` is provided, the `Deck` component will track view state changes from any attached `controller` using internal state, with `initialViewState` as its initial view state. This is the easiest way to control the camera.

If the `initialViewState` prop changes, the internally tracked view state will be updated to match the new "initial" view state.

Notes:

- The `onViewStateChange` callback will still be called, if provided.
- If the `viewState` prop is supplied by the application, the supplied `viewState` will always be used, overriding the `Deck` component's internal view state.
- In simple applications, use of the `initialViewState` prop can avoid the need to track the view state in the application.
- One drawback of using `initialViewState` for reactive/functional applications is that the `Deck` component becomes more stateful.

`effects` (Array)

The array of effects to be rendered. A lighting effect will be added if an empty array is supplied. Refer to effect's documentation to see details:

- LightingEffect
- PostProcessEffect

`_framebuffer` (Object)

(Experimental) Render to a custom frame buffer other than to screen.

`_animate` (Boolean)

- Default: `false`

(Experimental) Forces deck.gl to redraw layers every animation frame. Normally deck.gl layers are only redrawn if any change is detected.

Interaction Settings

`controller` (Function | Boolean | Object)

- Default: `null`

Options for viewport interactivity, e.g. pan, rotate and zoom with mouse, touch and keyboard. This is a shorthand for defining interaction with the `views` prop if you are using the default view (i.e. a single `MapView`).

```
new Deck({
  // ...
  views: [
    new MapView({
      controller: {touchRotate: true, doubleClickZoom: false}
    })
  ]
})
```

is equivalent to:

```
new Deck({
  // ...
  // views: undefined
  controller: {touchRotate: true, doubleClickZoom: false}
})
```

`controller` can be one of the following types:

- `null` or `false`: the viewport is not interactive.
- `true`: initiates the default controller of the default view - e.g. a MapController if `MapView` is used.
- `Controller` class (not instance): initiates the provided controller with default options. Must be a subclass of Controller.
- `Object`: controller options. This will be merged with the default controller options.
  - `controller.type`: the controller class, must be a subclass of Controller.
  - Other options supported by the controller type. Consult the documentation of Controller.

`getCursor` (Function)

- Default: `({isDragging}) => isDragging ? 'grabbing' : 'grab'`

A custom callback to retrieve the cursor type.

Receives arguments:

- `interactiveState` (Object)
  - `isDragging` (Boolean) - whether the pointer is down and moving
  - `isHovering` (Boolean) - whether the pointer is over a pickable object

Returns:

A valid CSS cursor string.

Remarks:

- It is worth noting that when supplying a custom image for the cursor icon, Chrome requires a fallback option to be supplied, otherwise the custom image will not be loaded; e.g. `getCursor={() => 'url(images/custom.png), auto'}`

`getTooltip` (Function)

Callback that takes a hovered-over point and renders a tooltip. If the prop is not specified, the tooltip is always hidden.

Receives arguments:

- `info` - the picking info describing the object being hovered.

Returns one of the following:

- `null` - the tooltip is hidden, with the CSS `display` property set to `none`.
- A string - the string is rendered in a tooltip with the default CSS styling described below.
- An object with the following fields:
  - `text` (String, optional) - Specifies the `innerText` attribute of the tooltip.
  - `html` (String, optional) - Specifies the `innerHTML` attribute of the tooltip. Note that this will override the specified `innerText`.
  - `className` (String, optional) - Class name to attach to the tooltip element. The element has the default class name of `deck-tooltip`.
  - `style` (Object, optional) - An object of CSS styles to apply to the tooltip element,

which can override the default styling.

By default, the tooltip has the following CSS style:

```
z-index: 1;
position: absolute;
color: #a0a7b4;
background-color: #29323c;
padding: 10px;
```

`pickingRadius` **(Number)**

Extra pixels around the pointer to include while picking. This is helpful when rendered objects are difficult to target, for example irregularly shaped icons, small moving circles or interaction by touch. Default `0`.

`touchAction` **(String)**

- Default: `none`.

Allow browser default touch actions. See hammer.js documentation.

By default, the deck canvas captures all touch interactions. This prop is useful for mobile applications to unblock default scrolling behavior. For example, use the combination `controller: {dragPan: false}` and `touchAction: 'pan-y'` to allow vertical page scroll when dragging over the canvas.

`eventRecognizerOptions` **(Object)**

- default: `{}`

Set options for gesture recognition. May contain the following fields:

- `pan` - an object that is Hammer.Pan options. This gesture is used for drag events.
- `pinch` - an object that is Hammer.Pinch options This gesture is used for two-finger touch events.
- `tripan` - an object that is Hammer.Pan options. This gesture is used for three-finger touch events.
- `tap` - an object that is Hammer.Tap options. This gesture is used for the `onClick` callback.

- `doubletap` - an object that is Hammer.Tap options. This gesture is used for double click events.

For example, the following setting makes panning less sensitive and clicking easier on mobile:

```
new Deck({
  // ...
  eventRecognizerOptions: isMobile ? {
    pan: {threshold: 10},
    tap: {threshold: 5}
  } : {}
})
```

`_pickable` **(Boolean)**

- Default: `true`

(Experimental) If set to `false`, force disables all picking features, disregarding the `pickable` prop set in any layer.

Event Callbacks

`onWebGLInitialized` **(Function)**

Called once the WebGL context has been initiated.

Receives arguments:

- `gl` - the WebGL context.

`onViewStateChange` **(Function)**

Called when the user has interacted with the deck.gl canvas, e.g. using mouse, touch or keyboard.

`onViewStateChange({viewState, interactionState, oldViewState})`

Receives arguments:

- `viewState` - An updated view state object.
- `interactionState` - Describes the interaction that invoked the view state change. May include the following fields:

  - `inTransition` (Boolean)
  - `isDragging` (Boolean)
  - `isPanning` (Boolean)
  - `isRotating` (Boolean)
  - `isZooming` (Boolean)
- `oldViewState` - The previous view state object.

Returns:

A view state object that is used to update `Deck`'s internally tracked view state (see `initialViewState`). This can be used to intercept and modify the view state before the camera updates, see add constraints to view state example.

If no value is returned, it's equivalent to `(viewState) => viewState`.

`onInteractionStateChange` (Function)

Called when the user has interacted with the deck.gl canvas, e.g. using mouse, touch or keyboard.

`onInteractionStateChange(interactionState)`

Receives arguments:

- `interactionState` - Describes the current interaction. May include the following fields:
  - `inTransition` (Boolean)
  - `isDragging` (Boolean)
  - `isPanning` (Boolean)
  - `isRotating` (Boolean)
  - `isZooming` (Boolean)

Note:

- `onInteractionStateChange` may be fired without `onViewStateChange`. For example, when the pointer is released at the end of a drag-pan, `isDragging` is reset to `false`, without the viewport's `longitude` and `latitude` changing.

`onHover` (Function)

Called when the pointer moves over the canvas.

Receives arguments:

- `info` - the picking info describing the object being hovered.
- `event` - the original gesture event

`onClick` (Function)

Called when clicking on the canvas.

Receives arguments:

- `info` - the picking info describing the object being clicked.
- `event` - the original gesture event

`onDragStart` (Function)

Called when the user starts dragging on the canvas.

Receives arguments:

- `info` - the picking info describing the object being dragged.
- `event` - the original gesture event

`onDrag` (Function)

Called when dragging the canvas.

Receives arguments:

- `info` - the picking info describing the object being dragged.
- `event` - the original gesture event

`onDragEnd` (Function)

Called when the user releases from dragging the canvas.

Receives arguments:

- `info` - the picking info describing the object being dragged.
- `event` - the original gesture event

`onLoad` (Function)

Called once after gl context and Deck components (`ViewManager`, `LayerManager`, etc) are created. It is safe to trigger viewport transitions after this event.

`onResize` (Function)

Called when the canvas resizes.

Receives arguments:

- `size`
  - `width` (Number) - the new width of the deck canvas, in client pixels
  - `height` (Number) - the new height of the deck canvas, in client pixels

`onBeforeRender` (Function)

Called just before the canvas rerenders.

Receives arguments:

- `gl` - the WebGL context.

`onAfterRender` (Function)

Called right after the canvas rerenders.

Receives arguments:

- `gl` - the WebGL context.

`onError` (Function)

- Default: `console.error`

Called if deck.gl encounters an error. By default, deck logs the error to console and attempt to continue rendering the rest of the scene. If this callback is set to `null`, errors are silently ignored.

Receives arguments:

- `error` (Error)
- `layer` (Layer?) - the layer where the error is originated, if applicable

`_onMetrics` (Function)

(Experimental) Called once every second with performance metrics.

Receives arguments:

- `metrics` - an object with fields specified here.

Methods

`finalize`

Frees all resources associated with this `Deck` instance.

`deck.finalize()`

`getCanvas`

Get the canvas element attached to this `Deck` instance.

`deck.getCanvas()`

Returns:

- Either an `HTMLCanvasElement` or `null` if one isn't assigned.

Notes:

- See the canvas prop for more information.

`setProps`

Updates (partial) properties.

```
deck.setProps({...});
```

Parameters:

- One or more properties to update, as described in the "Properties" section on this page.

`redraw`

Attempt to draw immediately, rather than waiting for the next draw cycle. By default, deck flushes all changes to the canvas on each animation frame. This behavior might cause the deck canvas to fall out of sync with other components if synchronous updates are required.

Redrawing frequently outside of rAF may cause performance problems. Only use this method if the render cycle must be managed manually.

```
deck.redraw(true);
```

Parameters:

- `force` (Boolean) - if `false`, only redraw if necessary (e.g. changes have been made to views or layers). If `true`, skip the check. Default `false`.

`pickObject`

Get the closest pickable and visible object at the given screen coordinate.

```
deck.pickObject({x, y, radius, layerIds})
```

Parameters:

- `x` (Number) - x position in pixels
- `y` (Number) - y position in pixels
- `radius` (Number, optional) - radius of tolerance in pixels. Default `0`.
- `layerIds` (Array, optional) - a list of layer ids to query from. If not specified, then all pickable and visible layers are queried.
- `unproject3D` (Boolean, optional) - if `true`, `info.coordinate` will be a 3D point by unprojecting the `x, y` screen coordinates onto the picked geometry. Default `false`.

Returns:

- a single `info` object, or `null` if nothing is found.

`pickMultipleObjects`

Performs deep picking. Finds all close pickable and visible object at the given screen coordinate, even if those objects are occluded by other objects.

```
deck.pickMultipleObjects({x, y, radius, layerIds, depth})
```

- `x` (Number) - x position in pixels
- `y` (Number) - y position in pixels
- `radius` (Number, optional) - radius of tolerance in pixels. Default `0`.
- `layerIds` (Array, optional) - a list of layer ids to query from. If not specified, then all pickable and visible layers are queried.
- `depth` - Specifies the max number of objects to return. Default `10`.
- `unproject3D` (Boolean, optional) - if `true`, `info.coordinate` will be a 3D point by unprojecting the `x, y` screen coordinates onto the picked geometry. Default `false`.

Returns:

- An array of `info` objects. The array will be empty if no object was picked.

Notes:

- Deep picking is implemented as a sequence of simpler picking operations and can have a performance impact. Should this become a concern, you can use the `depth` parameter to limit the number of matches that can be returned, and thus the maximum number of picking operations.

`pickObjects`

Get all pickable and visible objects within a bounding box.

```
deck.pickObjects({x, y, width, height, layerIds})
```

Parameters:

- `x` (Number) - left of the bounding box in pixels
- `y` (Number) - top of the bouding box in pixels
- `width` (Number, optional) - width of the bouding box in pixels. Default `1`.
- `height` (Number, optional) - height of the bouding box in pixels. Default `1`.
- `layerIds` (Array, optional) - a list of layer ids to query from. If not specified, then all pickable and visible layers are queried.

- `maxObjects` (Number, optional) - if specified, limits the number of objects that can be returned.

Returns:

- an array of unique `info` objects

Notes:

- The query methods are designed to quickly find objects by utilizing the picking buffer.
- The query methods offer more flexibility for developers to handle events compared to the built-in hover and click callbacks.

Member Variables

`isInitialized`

Flag indicating that the Deck instance has initialized its resources. It is safe to call public methods when `isInitialized` is `true`.

`metrics`

A map of various performance statistics for the last 60 frames of rendering. Metrics gathered in deck.gl are the following:

- `fps` - average number of frames rendered per second
- `updateAttributesTime` - time spent updating layer attributes
- `setPropsTime` - time spent setting deck properties
- `framesRedrawn` - number of times the scene was rendered
- `pickTime` - total time spent on picking operations
- `pickCount` - number of times a pick operation was performed
- `gpuTime` - total time spent on GPU processing
- `gpuTimePerFrame` - average time spent on GPU processing per frame
- `cpuTime` - total time spent on CPU processing
- `cpuTimePerFrame` - average time spent on CPU processing per frame
- `bufferMemory` - total GPU memory allocated for buffers
- `textureMemory` - total GPU memory allocated for textures

- `renderbufferMemory` - total GPU memory allocated for renderbuffers
- `gpuMemory` - total allocated GPU memory

Source

modules/core/src/lib/deck.ts

Edit this page