Layer Class

The `Layer` class is the base class of all deck.gl layers, and it provides a number of **base properties** available in all layers.

Static Members

`layerName` **(String, required)**

This static property should contain the name of the layer, typically the name of layer's class (it cannot reliably be autodeduced in minified code). It is used as the default Layer id as well as for debugging and profiling.

`defaultProps` **(Object, optional)**

All deck.gl layers define a `defaultProps` static member listing their props and default values. Using `defaultProps` improves both the code readability and performance during layer instance construction.

Constructor

```
new Layer(...props);
```

Parameters:

- `props` (Object) - `Layer` properties.

Notes:

- More than one property object can be supplied.
- Property objects will be merged with the same semantics as Object.assign, i.e. props in later objects will overwrite props earlier object.
- Every layer specifies default values for all its props, and these values will be used internally if that prop is not specified by the application in the Layer constructor.

Properties

Basic Properties

`id` **(String, optional)**

- Default: the layer class's `layerName` static property.

The `id` must be unique among all your layers at a given time. The default value of `id` is the `Layer`'s "name". If more than one instance of a specific type of layer exist at the same time, they must possess different `id` strings for deck.gl to properly distinguish them.

Remarks:

- `id` is used to match layers between rendering calls. deck.gl requires each layer to have a unique `id`. A default `id` is assigned based on layer type, which means if you are using more than one layer of the same type (e.g. two `ScatterplotLayer`s) you need to provide a custom `id` for at least one of them.
- A layer's name is defined by the `Layer.layerName` static member, or inferred from the constructor name (which is not robust in minified code).
- For sublayers (automatically generated by composite layers), a unique "composite layer id" is automatically generated by appending the sub layer's id to the parent layer's `id`, so there is no risk of `id` collisions for sublayers, as long as the application's layer names are unique.

`data` **(Iterable | String | Promise | AsyncIterable | Object, optional)**

- Default: `[]`

deck.gl layers typically expect `data` to be one of the following types:

- `Array`: a JavaScript array of data objects.
- An object that implements the iterable protocol, for example `Map` and `Set`. Represents a collection of data objects.
- Any non-iterable object that contains a `length` field. deck.gl will not attempt to interpret its format, but simply call each accessor `length` times. (See remark below)
- `String`: an URL pointing to the data source. deck.gl will attempt to fetch the remote content and parse it as JSON. The resulting object is then used as the value of the `data`

prop.

- `Promise`: the resolved value will be used as the value of the `data` prop.
- `AsyncIterable`: an async iterable object that yields data in batches. The default implementation expects each batch to be an array of data objects; one may change this behavior by supplying a custom `dataTransform` callback.

data.attributes

When using a non-iterable `data` object, the object may optionally contain a field `attributes`, if the application wishes to supply binary buffers directly to the layer. This use case is discussed in detail in the performance developer guide.

The keys in `data.attributes` correspond to the accessor name that the binary should replace, for example `getPosition`, `getColor`. See each layer's documentation for available accessor props.

Each value in `data.attributes` may be one of the following formats:

- luma.gl Buffer instance
- A typed array, which will be used to create a `Buffer`
- An object containing the following optional fields. For more information, see WebGL vertex attribute API.
  - `buffer` (Buffer)
  - `value` (TypedArray)
  - `type` (GLenum) - A WebGL data type, see vertexAttribPointer.
  - `size` (Number) - the number of elements per vertex attribute.
  - `offset` (Number) - offset of the first vertex attribute into the buffer, in bytes
  - `stride` (Number) - the offset between the beginning of consecutive vertex attributes, in bytes
  - `normalized` (Boolean) - whether data values should be normalized. Note that all color attributes in deck.gl layers are normalized by default.

Remarks

- Some layers may accept alternative data formats. For example, the GeoJsonLayer supports any valid GeoJSON object as `data`. These exceptions, if any, are documented in each layer's documentation.

- When an iterable value is passed to `data`, every accessor function receives the current element as the first argument. When a non-iterable value (any object with a `length` field) is passed to `data`, the accessor functions are responsible of interpreting the data format. The latter is often used with binary inputs. Read about this in accessors.
- The automatic URL loading support is intended as a convenience for simple use cases. There are many limitations, including control over request parameters and headers, ability to parse non-JSON data, control of post-processing, CORS support etc. In such cases, it is always possible to load data using traditional JavaScript techniques and rerender the layer with the resulting data array.
- For `Layer` writers: Even though applications can pass in url string as `data` props, deck.gl makes sure that the layers never see `data` props containing `String` values. Layers will only see the default empty array value for the `data` prop until the data is loaded, at which point it will be updated with the loaded array.

`visible` (Boolean, optional)

- Default: `true`

Whether the layer is visible. Under most circumstances, using `visible` prop to control the visibility of layers are recommended over doing conditional rendering. Compare:

```
const layers = [
  new MyLayer({data: ..., visible: showMyLayer})
];
```

with

```
const layers = [];
if (showMyLayer) {
  layers.push(new MyLayer({data: ...}));
}
```

In the second example (conditional rendering) the layer state will be destroyed and regenerated every time the `showMyLayer` flag changes.

`opacity` (Number, optional)

- Default: `1`

The opacity of the layer.

Remarks:

- deck.gl automatically applies gamma to the opacity in an attempt to make opacity changes appear linear (i.e. the perceived opacity is visually proportional to the value of the prop).
- While it is a recommended convention that all deck.gl layers should support the `opacity` prop, it is up to each layer's fragment shader to properly implement support for opacity.

`extensions` **(Array, optional)**

Add additional functionalities to this layer. See the list of available extensions.

`onError` **(Function, optional)**

Called when this layer encounters an error, with the following arguments:

- `error` - a JavaScript `Error` object.

If this callback is supplied and returns `true`, the error is marked as handled and will not bubble up to the `onError` callback of the `Deck` instance.

Interaction Properties

Layers can be interacted with using these properties.

`pickable` **(Boolean, optional)**

- Default: `false`

Whether the layer responds to mouse pointer picking events.

`onHover` **(Function, optional)**

This callback will be called when the mouse enters/leaves an object of this deck.gl layer with the following parameters:

- `info`
- `event` - the source event

If this callback returns a truthy value, the `hover` event is marked as handled and will not

bubble up to the `onHover` callback of the `DeckGL` canvas.

Requires `pickable` to be true.

`onClick` (Function, optional)

This callback will be called when the mouse clicks over an object of this deck.gl layer with the following parameters:

- `info`
- `event` - the source event

If this callback returns a truthy value, the `click` event is marked as handled and will not bubble up to the `onClick` callback of the `DeckGL` canvas.

Requires `pickable` to be true.

`onDragStart` (Function, optional)

This callback will be called when the mouse starts dragging an object of this deck.gl layer with the following parameters:

- `info`
- `event` - the source event

If this callback returns a truthy value, the `dragstart` event is marked as handled and will not bubble up to the `onDragStart` callback of the `DeckGL` canvas.

Requires `pickable` to be true.

`onDrag` (Function, optional)

This callback will be called when the mouse drags an object of this deck.gl layer with the following parameters:

- `info`
- `event` - the source event

If this callback returns a truthy value, the `drag` event is marked as handled and will not bubble up to the `onDrag` callback of the `DeckGL` canvas.

Requires `pickable` to be true.

`onDragEnd` (Function, optional)

This callback will be called when the mouse releases an object of this deck.gl layer with the following parameters:

- `info`
- `event` - the source event

If this callback returns a truthy value, the `dragend` event is marked as handled and will not bubble up to the `onDragEnd` callback of the `DeckGL` canvas.

Requires `pickable` to be true.

`highlightColor` (Array|Function, optional)

- Default: `[0, 0, 128, 128]`

RGBA color to blend with the highlighted object (either the hovered over object if `autoHighlight: true`, or the object at the index specified by `highlightedObjectIndex`). When the value is a 3 component (RGB) array, a default alpha of 255 is applied.

- If an array is supplied, it is used for the object that is currently highlighted.
- If a function is supplied, it is called with a pickingInfo object when the hovered object changes. The return value is used as the highlight color for the picked object. Only works with `autoHighlight: true`.

`highlightedObjectIndex` (Number|null, optional)

- Default: `null`

The index of the object in `data` to highlight. If specified, overrides the effect of `autoHighlight`.

When set to an integer that corresponds to an object, the object will be highlighted with `highlightColor`. When set to an integer that does not correspond to an object (e.g. `-1`), nothing is highlighted.

`autoHighlight` (Boolean, optional)

- Default: `false`

When true, current object pointed by mouse pointer (when hovered over) is highlighted with `highlightColor`.

Requires `pickable` to be true.

Coordinate System Properties

Normally only used when the application wants to work with coordinates that are not Web Mercator projected longitudes/latitudes.

`coordinateSystem` (Number, optional)

- Default: `COORDINATE_SYSTEM.DEFAULT`

Specifies how layer positions and offsets should be geographically interpreted. One of:

- `COORDINATE_SYSTEM.CARTESIAN`
- `COORDINATE_SYSTEM.LNGLAT`
- `COORDINATE_SYSTEM.METER_OFFSETS`
- `COORDINATE_SYSTEM.LNGLAT_OFFSETS`

The default is to interpret positions as latitude and longitude, however it is also possible to interpret positions as meter offsets added to projection center specified by the `coordinateOrigin` prop.

See the article on Coordinate Systems for details.

`coordinateOrigin` ([Number, Number, Number], optional)

- Default: `[0, 0, 0]`

Specifies a reference point for coordinates that are expressed as offsets. Used when the `coordinateSystem` is set to one of the following modes:

- `COORDINATE_SYSTEM.CARTESIAN` (optional)
- `COORDINATE_SYSTEM.METER_OFFSETS` (required)
- `COORDINATE_SYSTEM.LNGLAT_OFFSETS` (required)

See the article on Coordinate Systems for details.

`wrapLongitude` **(Boolean, optional)**

- Default: `false`

Normalizes geometry longitudes. Only works with the `LNGLAT` coordinate system.

When enabled on `PathLayer`, `PolygonLayer` and `GeoJsonLayer`, the paths/polygons are interpretted such that the connection between any two neighboring vertices is drawn on the shorter side of the world, and split into two if it crosses the 180th meridian. Note that this introduces CPU overhead at runtime. When working with static data, it is recommend that you preprocess the features offline instead of using this option.

When enabled on layers with position pairs, such as `LineLayer` and `ArcLayer`, they will draw the shortest path between source and target positions. If the shortest path crosses the 180th meridian, it is split into two segments.

When enabled on other layers, the effect is applied by moving the anchor point into the `[-180, 180]` range.

`modelMatrix` **(Number[16], optional)**

An optional 4x4 matrix that is multiplied into the affine projection matrices used by shader `project` GLSL function and the Viewport's `project` and `unproject` JavaScript function.

Allows local coordinate system transformations to be applied to a layer, which is useful when composing data from multiple sources that use different coordinate systems.

Note that the matrix projection is applied after the non-linear mercator projection calculations are resolved, so be careful when using model matrices with longitude/latitude encoded coordinates. They normally work best with non-mercator viewports or meter offset based mercator layers.

Data Properties

There are a number of additional properties that provide extra control over data iteration, comparison and update.

`dataComparator` **(Function, optional)**

This prop causes the `data` prop to be compared using a custom comparison function. The comparison function is called with the old data and the new data objects, and is expected to return true if they compare equally.

Used to override the default shallow comparison of the `data` object.

As an illustration, the app could set this to e.g. 'lodash.isequal', enabling deep comparison of the data structure. This particular examples would obviously have considerable performance impact and should only be used as a temporary solution for small data sets until the application can be refactored to avoid the need.

The function receives two arguments:

- `newData` - the new data prop
- `oldData` - the data prop before the update

And should return `true` if the two values are considered equal (no update needed), or `false` if the data has changed.

`dataTransform` (Function, optional)

Executed to manipulate remote data when it's fetched. This callback applies when `data` is assigned a value that is either string (URL), Promise or an async iterable.

The function receives two arguments:

- `data` - the newly fetched data
- `previousData` - (only available when `data` is an async iterable) the previously loaded data. If `dataTransform` is not supplied, the new data chunk is appended to the previous data array.

And must return a valid `data` object: an array, an iterable or a non-iterable object that contains a `length` field. See documentation of the `data` prop for details.

`_dataDiff` (Function, optional) Experimental

This function is called when the data changes, either shallowly when `dataComparator` is not supplied or because `dataComparator` returns `false`, to retrieve the indices of the changed objects. By default, when data changes, the attributes of all objects are recalculated. If this prop is supplied, only the attributes of the specified objects will be updated. This can lead to significant performance improvement if a few rows in a large data table need to change

frequently:

```
let data = [...];

function updateData(index, item) {
  // make a shallow copy
  data = data.slice();
  data.splice(index, 1, item);
  const layer = new ScatterplotLayer({
    data,
    // Only update the attributes at `index``
    _dataDiff: (newData, oldData) => [{startRow: index, endRow: index + 1}],
    ...
  });
  deck.setProps({layers: [layer]});
}
```

The function receives two arguments:

- `newData` - the new data prop
- `oldData` - the data prop before the update

And is expected to return an array of "ranges", in the form of `{startRow, endRow}`:

- `startRow` (Number) - the beginning index of a changed chunk in the new data.
- `endRow` (Number) - the end index of a changed chunk in the new data (excluded).

This feature is experimental and intended for advanced use cases. Note that it only rewrites part of a buffer, not remove or insert, therefore the user of this prop is responsible of making sure that all the unchanged objects remain at the same indices between `oldData` and `newData`. This becomes trickier when dealing with data of dynamic lengths, for example `PathLayer`, `PolygonLayer` and `GeoJsonLayer`. Generally speaking, it is not recommended to use this feature when the count of vertices in the paths/polygons may change.

`positionFormat` (String, optional)

One of `'XYZ'`, `'XY'`.

This prop is currently only effective in `PathLayer`, `SolidPolygonLayer` and `PolygonLayer`.

Default `'XYZ'`.

`colorFormat` **(String, optional)**

One of `'RGBA'`, `'RGB'`.

Setting it to `'RGB'` will make the layer ignore the alpha channel of the colors returned by accessors, and instead assume all objects to be opaque. The layer's overall transparency controlled by `opacity` is still applied.

Default `'RGBA'`.

`numInstances` **(Number, optional)**

deck.gl automatically derives the number of drawing instances from the `data` prop by counting the number of objects in `data`. However, the developer might want to manually override it using this prop.

`updateTriggers` **(Object, optional)**

Accessors such as `getColor` and `getPosition` are called to retrieve colors and positions when a layer is first added. From then on, to maximize performance, deck.gl does not recalculate colors or positions unless the `data` prop changes by shallow comparison.

Sometimes `data` remains the same, but the outcome of an accessor has changed. In the following example, this is caused by changes in the external values `maleColor` and `femaleColor`:

```
const layer = new ScatterplotLayer({
  ... // Other props
  getFillColor: d => d.male ? maleColor : femaleColor
});
```

In this case, you need to explicitly inform deck.gl to re-evaluate `getFillColor` for all data items. You do so by defining `updateTriggers`:

```
const layer = new ScatterplotLayer({
  ... // Other props
  getFillColor: d => d.male ? maleColor : femaleColor,
  updateTriggers: {
      getFillColor: [maleColor, femaleColor]
  }
});
```

`updateTriggers` expect an object whose keys are names of accessor props of this layer, and values are one or more variables that affect the output of the accessors.

For example, `updateTriggers.getFillColor` is a list of variables that affect the output of `getFillColor`. If either value in the array changes, all attributes that depend on `getFillColor` will be updated. The variables may be numbers, strings, objects or functions. During each rendering cycle, deck.gl shallow-compares them with the previous values.

Note:

- change of the `data` prop has higher priority than the `updateTriggers`. If the app supplies a new `data` object, then all attributes will be automatically updated, even if the updateTriggers have not changed. To block excessive attribute updates, set the `dataComparator` prop.

`loaders` (Array, optional)

- Default: `[]`

An array of loaders.gl loaders.

Out of the box, deck.gl supports loading JSON and image files (with ImageLoader), and specific layer implementation such as `MVTLayer`, `TerrainLayer` and `ScenegraphLayer` also include loaders for their targeted formats.

Support for addtional formats can be added by adding loaders to the `loaders` prop. For example, to load data from CSV files:

```
import {CSVLoader} from '@loaders.gl/csv';

const layer = new HexagonLayer({
  data: './data.csv',
  loaders: [CSVLoader]
  ...
});
```

`loadOptions` (Object, optional)

- Default: `null`

Layers use loaders.gl to load data and other resources. `loadOptions` is an object to customize

the behavior of any loader used during fetching and parsing. The object may contain any of the top-level options, as well as loader-specific options targeting a particular format.

Layers may also include specialized loaders for their own use case, such as images, 3d-tiles, etc. See the documentation of each layer for additional load options that are available.

Find usage examples in the data loading guide.

`fetch` (Function, optional)

Called to fetch and parse content from URLs.

The function receives the following arguments:

- `url` (String) - the URL to fetch
- `context` (Object)
  - `layer` (Layer) - the current layer
  - `propName` (String) - the name of the prop that is making the request
  - `loaders` (Array?) - an array of loaders.gl loaders to parse this request with, default to `props.loaders`.
  - `loadOptions` (Object?) - loader options for this request, default to `props.loadOptions`.
  - `signal` (AbortSignal?) - the signal to abort the request

The function is expected to return a Promise that resolves to loaded data.

- Default: `load(url, loaders, loadOptions)`

`onDataLoad` (Function, optional)

Called when remote data is fully loaded. This callback applies when `data` is assigned a value that is either string (URL), Promise or an async iterable.

The function receives two arguments:

- `value` - the loaded data
- `context` (Object)
  - `layer` - the current layer

Render Properties

`parameters` **(Object, optional)**

The `parameters` allows applications to specify values for WebGL parameters such as blending mode, depth testing etc. Any `parameters` will be applied temporarily while rendering this layer only.

To get access to static parameter values, applications can `import GL from '@luma.gl/constants'`. Please refer to the luma.gl setParameters API for documentation on supported parameters and values.

`getPolygonOffset` **(Function, optional)**

- Default: `({layerIndex}) => [0, -layerIndex * 100]`

When multiple layers are rendered on the same plane, z-fighting may create undesirable artifacts. To improve the visual quality of composition, deck.gl allows layers to use `gl.polygonOffset` to apply an offset to its depth. By default, each layer is offset a small amount by its index so that layers are cleanly stacked from bottom to top.

This accessor takes a single parameter `uniform` - an object that contains the current render uniforms, and returns an array of two numbers `factor` and `units`. Negative values pull layer towards the camera, and positive values push layer away from the camera. For more information, refer to the documentation and FAQ.

If the accessor is assigned a falsy value, polygon offset will be set to `[0, 0]`.

Remarks:

- While this feature helps mitigate z-fighting, at close up zoom levels the issue might return because of the precision error of 32-bit projection matrices. Try set the `fp64` prop to `true` in this case.

`transitions` **(Object, optional)**

- Default: `{}`

When creating layers, animation can be enabled by supplying an `transitions` prop. Animation parameters are defined per attribute by using attribute names or accessor names as keys:

```
new Layer({
```

```
  transitions: {
    getPositions: 600,
    getColors: {
      duration: 300,
      easing: d3.easeCubicInOut,
      enter: value => [value[0], value[1], value[2], 0] // fade in
    },
    getRadius: {
      type: 'spring',
      stiffness: 0.01,
      damping: 0.15,
      enter: value => [0] // grow from size 0
    }
  }
});
```

Each accessor name is mapped to an object that is the transition setting. The object may contain the following fields:

Key	Type	Default	Description
`type`	`String`	`'interpolation'`	Type of the transition (either `'interpolation'` or `'spring'`)
`enter`	`Function`	APPEARANCE (`value => value`)	Callback to get the value that the entering vertices are transitioning from. See notes below
`onStart`	`Function`	`null`	Callback when the transition is started
`onEnd`	`Function`	`null`	Callback when the transition is done
`onInterrupt`	`Function`	`null`	Callback when the transition is interrupted

Additional fields for `type: 'interpolation'`:

Key	Type	Default	Description

Key	Type	Default	Description
`duration`	`Number`	`0`	Duration of the transition animation, in milliseconds
`easing`	`Function`	LINEAR (`t => t`)	Easing function that maps a value from [0, 1] to [0, 1], see http://easings.net/

Additional fields for `type: 'spring'`:

Key	Type	Default	Description
`stiffness`	`Number`	`0.05`	"Tension" factor for the spring
`damping`	`Number`	`0.5`	"Friction" factor that counteracts the spring's acceleration

Notes:

- As a shorthand, if an accessor key maps to a number rather than an object, then the number is assigned to the `duration` parameter, and an `interpolation` transition is used.
- Attribute transition is performed between the values at the same index. If the new data is larger, `enter` callback is called for each new vertex to backfill the values to transition from.
- `enter` should return the value to transition from. for the current vertex. It receives two arguments:
  - `toValue` (TypedArray) - the new value to transition to, for the current vertex
  - `fromChunk` (Array | TypedArray) - the existing value to transition from, for the chunk that the current vertex belongs to. A "chunk" is a group of vertices that help the callback determine the context of this transition. For most layers, all objects are in one chunk. For PathLayer and PolygonLayer, each path/polygon is a chunk.

Members

> Layer members are designed to support the creation of new layers or layer sub-classing and are NOT intended to be used by applications.

`context` **(Object)**

The context object stores information that are shared by all layers.

- `gl` (WebGLRenderingContext) - WebGL context of the current canvas.
- `viewport` (Viewport) - The current viewport
- `deck` (Deck) - The current deck.gl instance

`state` **(Object)**

The state object allows a layer to store persistent information cross rendering cycles.

- `attributeManager` (AttributeManager) - The attribute manager of this layer.

`props` **(Object)**

Properties of this layer.

`isLoaded`

`true` if all asynchronous assets are loaded.

`parent`

A `Layer` instance if this layer is rendered by a CompositeLayer

Methods

> Layer methods are designed to support the creation of new layers or layer sub-classing and are NOT intended to be called by applications.

General Methods

`clone`

```
const updatedLayer = layer.clone(overrideProps);
```

Create a copy of this layer, optionally change some prop values.

Arguments:

- `overrideProps` (Object, optional) - layer props to update.

`setState`

Used to update the layers `state` object. Calling this method will also cause the layer to rerender.

`setModuleParameters`

Used to update the settings of shader modules.

`raiseError`

Used to propagate errors thrown in layers to the deck.gl error handling system. This prevents rendering from being interrupted by exceptions in layers.

`raiseError(error, message)`

- `error` (Error) - a JavaScript Error object
- `message` (String, optional) - additional contextual description to amend the error message with

Layer Lifecycle Methods

For more information about when these methods are called, see layer lifecycle.

`initializeState`

This method is called only once for each layer to set up the initial state.

`initializeState(context)`

- `context` - The layer context is supplied as a parameter
  - `context.gl` (`WebGLRenderingContext`) - gl context
  - ...

deck.gl will already have created the `state` object at this time, and added the `gl` context and the `attributeManager` state.

`shouldUpdateState`

Called during each rendering cycle when layer properties or context has been updated and

before layers are drawn.

`shouldUpdateState({props, oldProps, context, changeFlags})`

Parameters:

- `props` (Object) - Layer properties from the current rendering cycle.
- `oldProps` (Object) - Layer properties from the previous rendering cycle.
- `context` (Object) - Layer context from the current rendering cycle.
- `changeFlags`:
  - an object that contains the following boolean flags:
  - `dataChanged`, `propsChanged`, `viewportChanged`, `somethingChanged`, `propsOrDataChanged`, `stateChanged`, `updateTriggersChanged`, `viewportChanged`

Returns:

- `true` this layer needs to be updated.

Remarks:

- Prop change is determined by shallow comparison.
- Data change is determined by shallow comparison of `props.data` unless `dataComparator` is supplied.
- The default implementation returns `true` if any change has been detected in data or props, but **ignores viewport changes**.

`updateState`

Called when a layer needs to be updated.

`updateState({props, oldProps, context, changeFlags})`

Parameters:

- `props` (Object) - Layer properties from the current rendering cycle.
- `oldProps` (Object) - Layer properties from the previous rendering cycle.
- `context` (Object) - Layer context from the current rendering cycle.
- `changeFlags`:
  - an object that contains the following boolean flags:

    - `dataChanged`, `propsChanged`, `viewportChanged`, `somethingChanged`, `propsOrDataChanged`, `stateChanged`, `updateTriggersChanged`, `viewportChanged`

The default implementation will invalidate all attributeManager attributes if the `data` prop has changed.

`draw`

Called on a layer to render to the WebGL canvas.

`draw({moduleParameters, uniforms, ..., context})`

Parameters:

- `uniforms`:
- an object that contains all the default unforms to be passed to the shaders.
- `context` - The layer context is supplied as a parameter
    - `context.gl` (`WebGLRenderingContext`) - gl context
    - ...

The default implementation looks for a variable `model` in the layer's state (which is expected to be an instance of the luma.gl `Model` class) and calls `draw` on that model with the parameters.

`getPickingInfo`

Called when a layer is being hovered or clicked, before any user callbacks are called. The layer can override or add additional fields to the `info` object that will be passed to the callbacks.

Parameters:

- `pickParams` (Object)
    - `pickParams.info` (Object) - The current `info` object. By default it contains the following fields:
        - `x` (Number) - Mouse position x relative to the viewport.
        - `y` (Number) - Mouse position y relative to the viewport.

        - `coordinate` ([Number, Number]) - Mouse position in world coordinates. Only applies if `coordinateSystem` is `COORDINATE_SYSTEM.LNGLAT`.
        - `color` (Number[4]) - The color of the pixel that is being picked. It represents a "picking color" that is encoded by `layer.encodePickingColor()`.
        - `index` (Number) - The index of the object that is being picked. It is the returned value of `layer.decodePickingColor()`.
        - `picked` (Boolean) - `true` if `index` is not `-1`.
    - `pickParams.mode` (String) - One of `hover` and `click`

Returns:

- An `info` object with optional fields about what was picked. This object will be passed to the layer's `onHover` or `onClick` callbacks.
- `null`, if the corresponding event should cancelled with no callback functions called.

The default implementation populates the `info.object` field with the `info.index` element from the layer's `data` prop.

`finalizeState`

This method is called before the layer is being removed. A layer should release all resources created during its life span.

Layer Projection Methods

While most projection is handled "automatically" in the layers vertex shader, it is occasionally useful to be able to work in the projected coordinates in JavaScript while calculating uniforms etc.

`project`

Projects a map coordinate to screen coordinate, using the current viewport settings and the current coordinate system.

Parameters:

- `coordinates` (Array) - `[x, y, z]` in this layer's coordinate system.

Returns:

- A screen coordinates array `[x, y]` or `[x, y, z]` if an altitude was given.

`unproject`

Unprojects a screen coordinate using the current viewport settings.

Parameters:

- `pixels` (Array) - `[x, y, z]` Passing a `z` is optional.

Returns:

- A map coordinates array `[lng, lat]` or `[lng, lat, altitude]` if a `z` was given.

`projectPosition`

Projects a map coordinate to world coordinate using the current viewport settings and the current coordinate system. Can be useful to calculate world space angle and distances.

Parameters:

- `coordinates` (Array) - `[x, y, z]` in this layer's coordinate system.
- `params` (Object, optional) - Projection parameters override. It accepts the following fields:
  - `viewport` (Viewport) - defaults to `this.context.viewport`.
  - `modelMatrix` (Matrix4) - defaults to `this.props.modelMatrix`.
  - `coordinateOrigin` ([Number, Number, Number]) - defaults to `this.props.coordinateOrigin`.
  - `coordinateSystem` (Number) - defaults to `this.props.coordinateSystem`.
  - `fromCoordinateOrigin` ([Number, Number, Number]) - defaults to passed `coordinateOrigin`.
  - `fromCoordinateSystem` (Number) - defaults to passed `coordinateSystem`.

Returns:

- World coordinates in `[x, y]`.

Layer Picking Methods

For the usage of these methods, see how picking works.

`decodePickingColor`

Converts a color to a "sub-feature index" number. This color is encoded by the `layer.encodePickingColor()` method.

Parameters:

- `color` (Array) - The color to be decoded in `[r, g, b]`.

Returns:

- A number representing the index of the feature. The null picking color (See `Layer.nullPickingColor`) will be decoded as -1.

Note:

- The null picking color is returned when a pixel is picked that is not covered by the layer, or when the layer has selected to render a pixel using the null picking color to make it unpickable.

`encodePickingColor`

Converts a "sub-feature index" number to a color. This color can later be decoded by the `layer.decodePickingColor()` method.

Parameters:

- `index` (Integer) - The index to be encoded.

Returns:

- An array of `[r, g, b]`.

To get a color that does not correspond to any sub-feature, use `layer.nullPickingColor()`.

Notes:

- indices to be encoded must be integers larger than or equal to 0.
- Picking colors are 24 bit values and can thus encode up to 16 million indices.

`nullPickingColor`

Returns:

- a "null" picking color which is equal the color of pixels not covered by the layer. This color is guaranteed not to match any index value greater than or equal to zero.

Source

modules/core/src/lib/layer.ts

Edit this page