Deck.gl

Lorenzo Adreani

Corso di: Big Data Architectures

Prof. Paolo Nesi

Dep DINFO, University of Florence

Via S. Marta 3, 50139, Firenze, Italy

DISIT Lab, Sistemi Distribuiti e Tecnologie Internet

http://www.disit.dinfo.unifi.it/ , Https://www.disit.org

paolo.nesi@unifi.it http://www.disit.dinfo.unifi.it/nesi

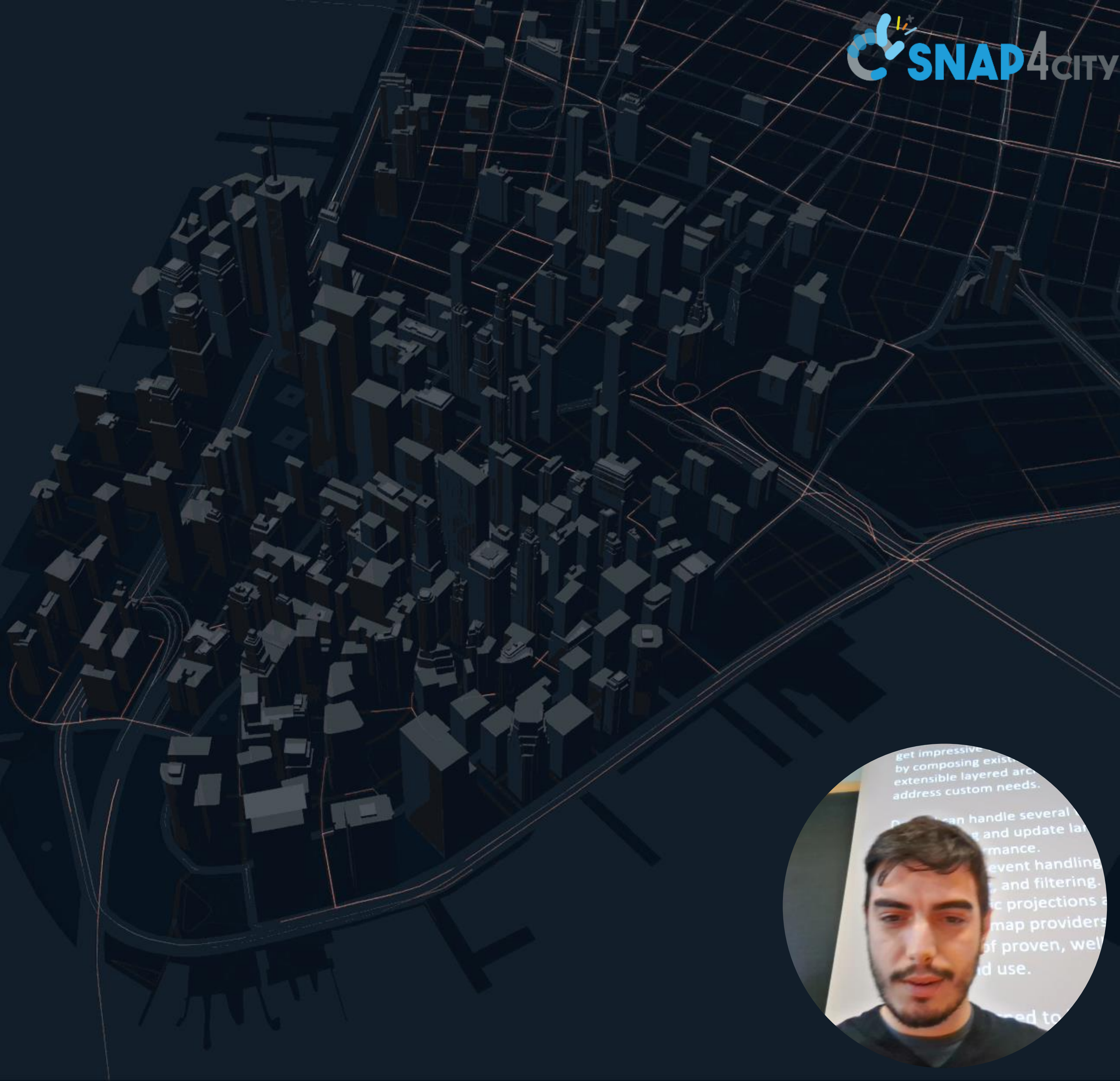
Deck.gl

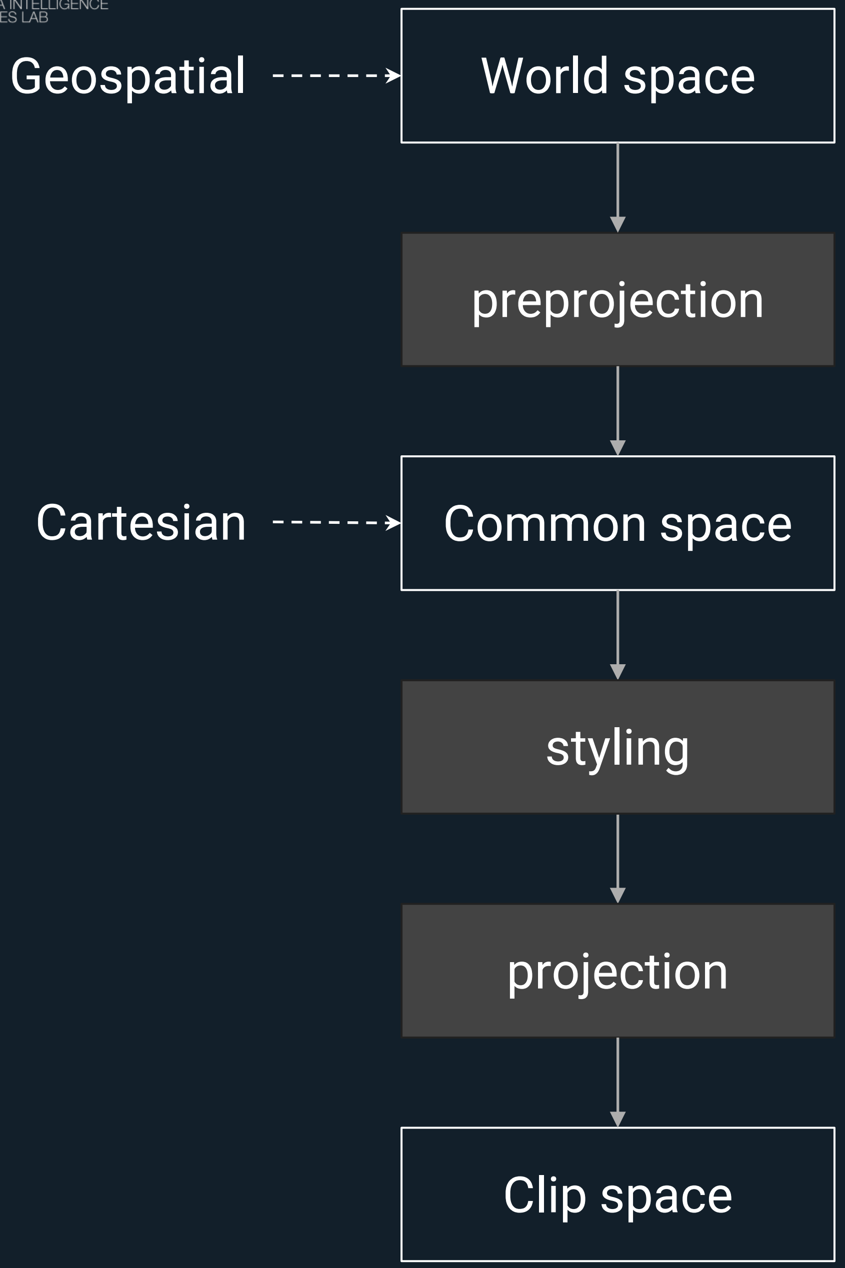
Deck.gl is a free and opensource library designed to offer high-performance for 3D WebGL-based visualization of large data sets. Users can quickly get impressive visual results with minimal effort by composing existing layers or exploiting the extensible layered architecture of Deck.gl to address custom needs.

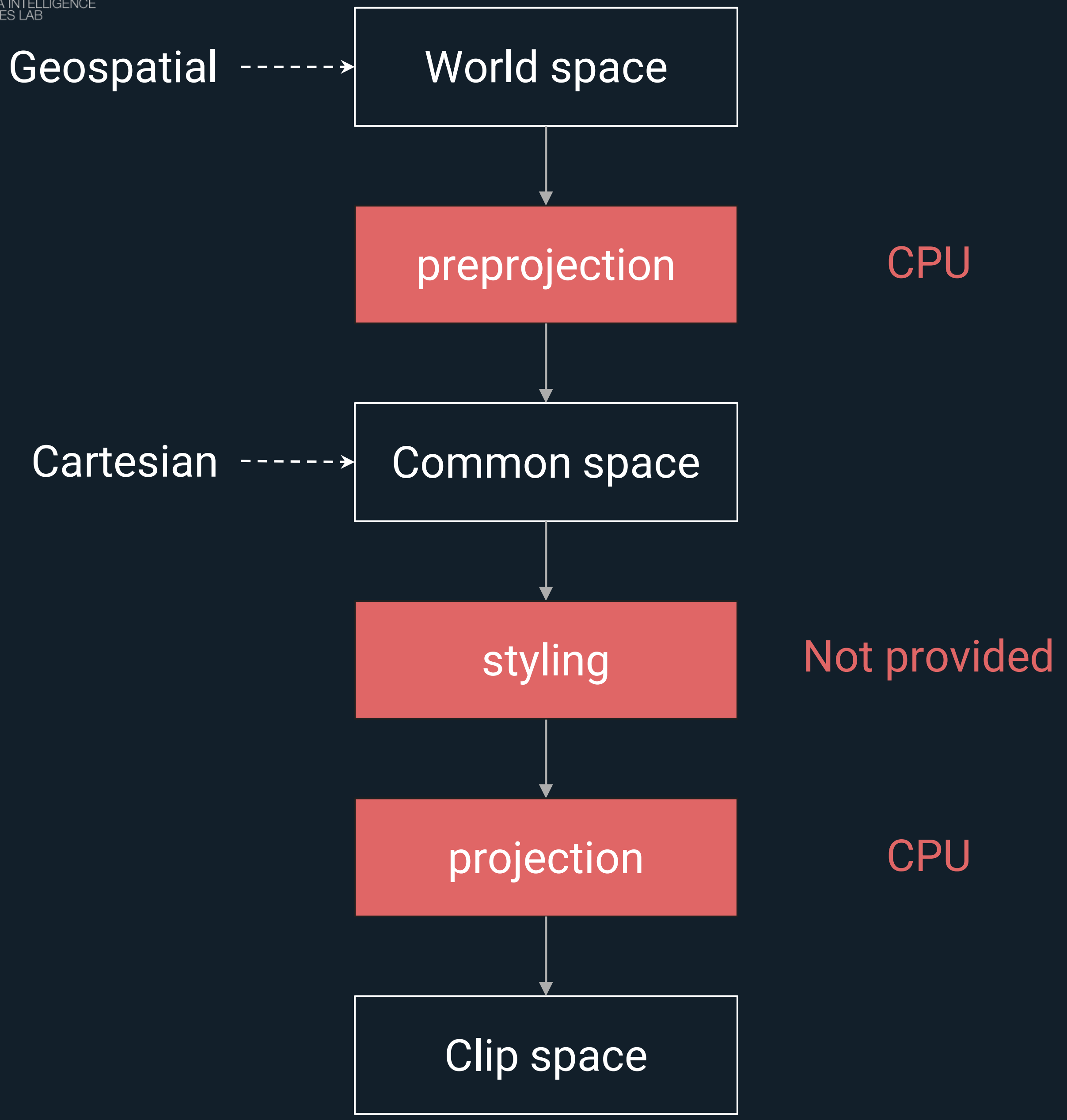
Deck.gl can handle several tasks out of the box:

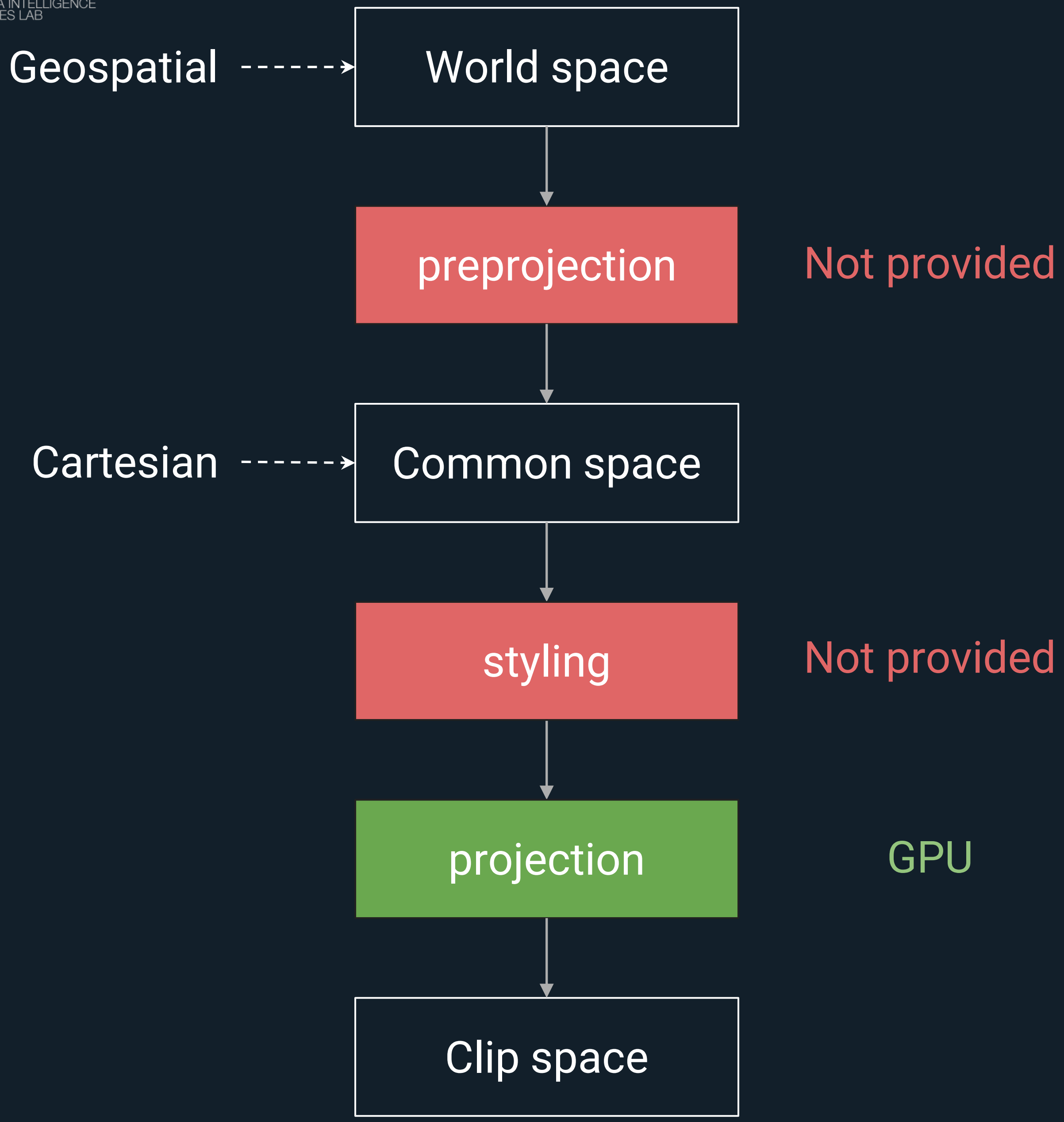
- Rendering and update large data sets with high-performance.
- Interactive event handling such as picking, highlighting, and filtering.
- Cartographic projections and integration with major basemap providers
- A catalog of proven, well-tested layers easy to deploy and use.

Deck.gl is designed to be highly customizable.



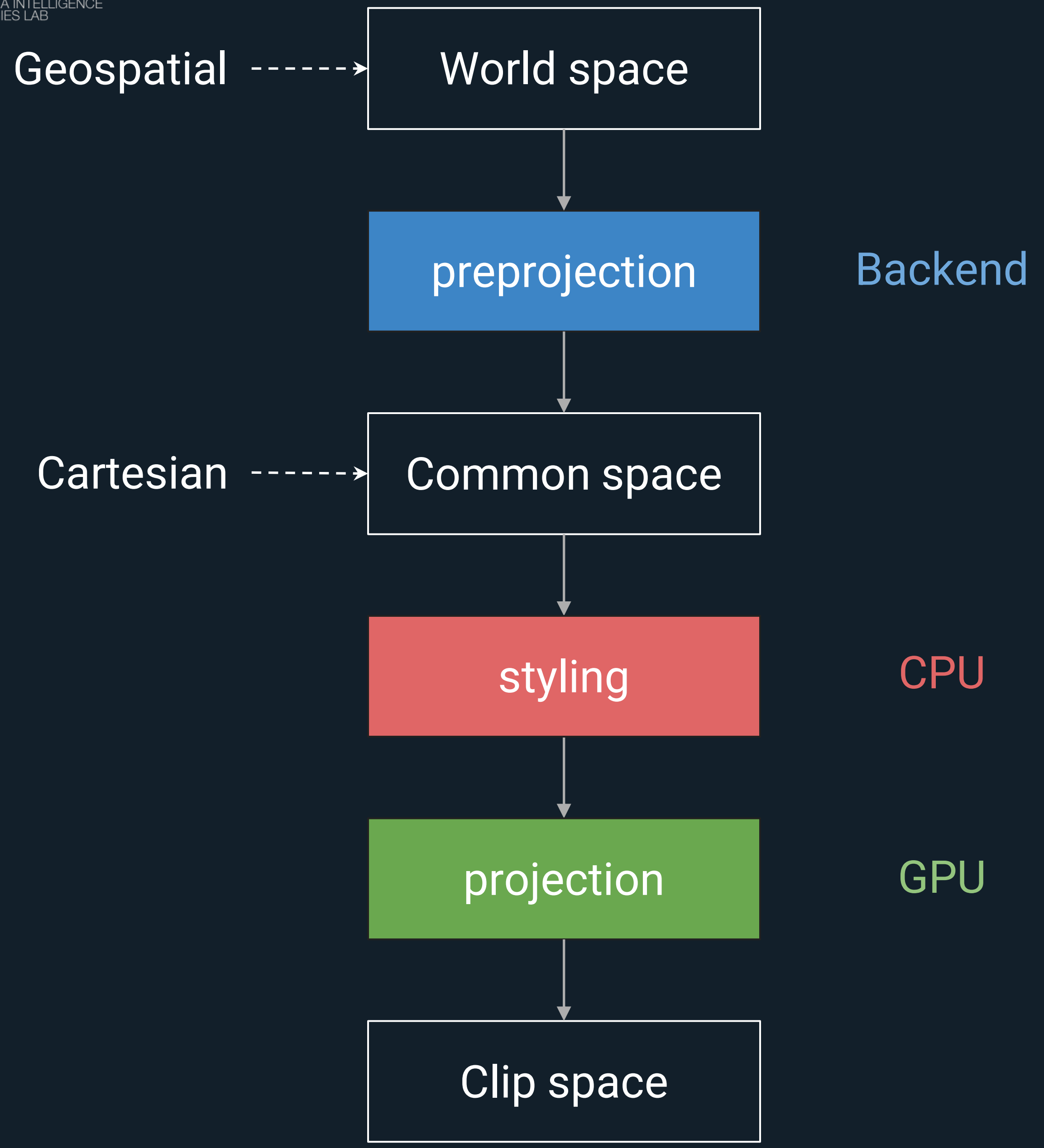

Geospatial
World space
preprojection
Cartesian
Common space
styling
projection
Clip space

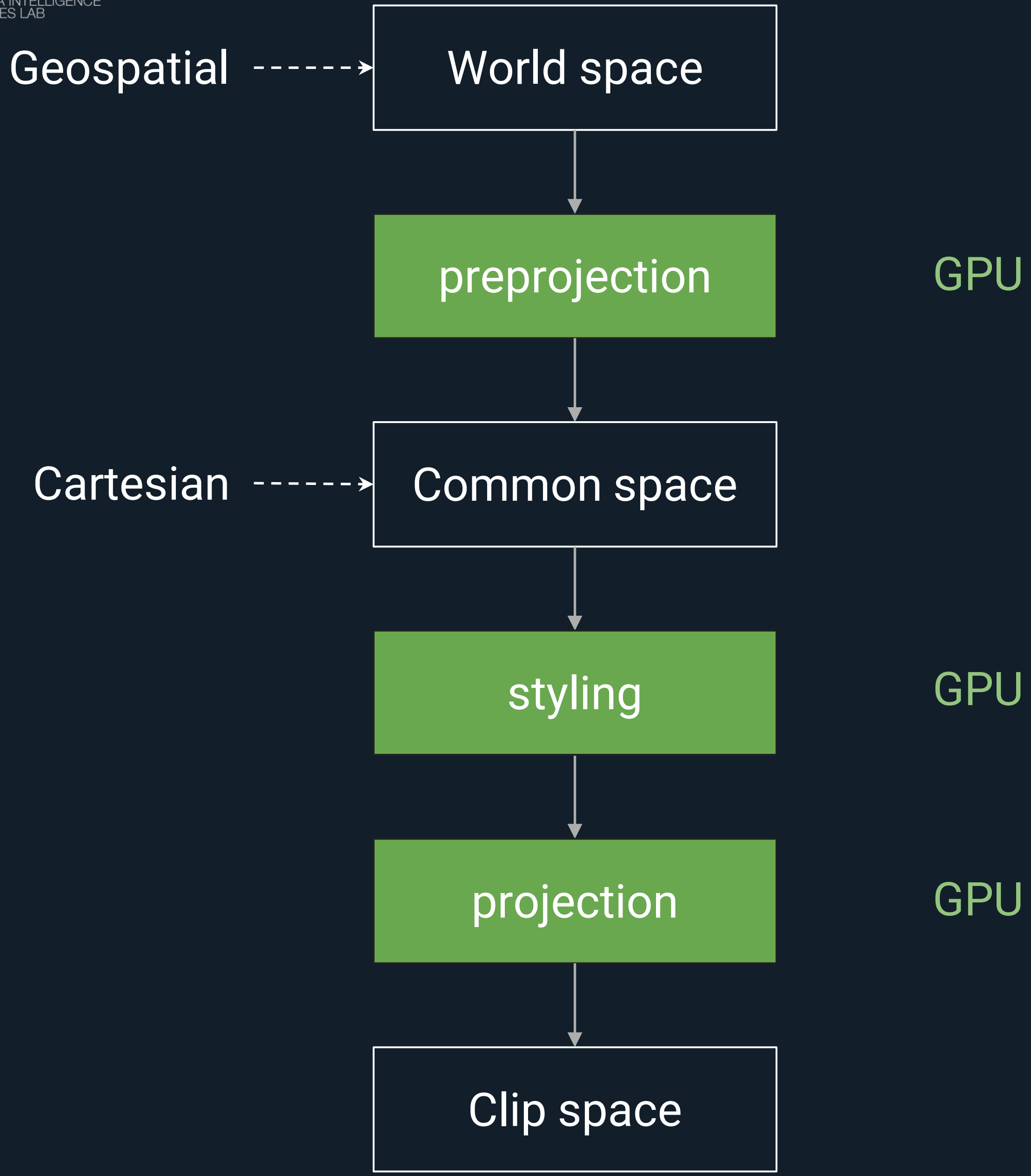

Geospatial
World space
preprojection
CPU
Cartesian
Common space
styling
Not provided
projection
CPU
Clip space



Geospatial
World space
preprojection
Not provided
Cartesian
Common space
styling
Not provided
projection
GPU
Clip space

Stock Wel


Geospatial
World space
preprojection
Backend
Cartesian
Common space
styling
CPU
projection
GPU
Clip space


Geospatial
World space
preprojection
GPU
Cartesian
Common space
styling
GPU
projection
GPU
Clip space

Aggregation times

Points	CPU Aggregation	GPU Aggregation	Improvement
25K	535 iterations/s	359 iterations/s	(slower)
100K	119 iterations/s	437 iterations/s	2.7x
1M	12.7 iterations/s	158 iterations/s	11x

Layers

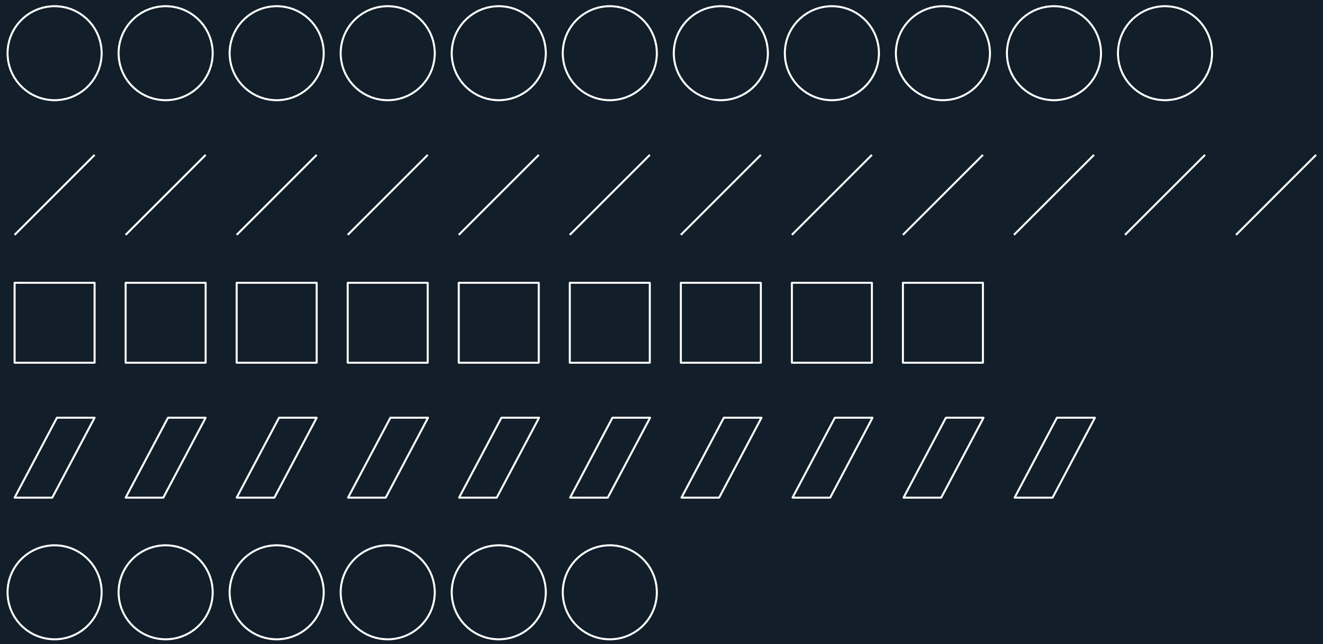
Layer (scatterplot) draw()

Layer (path) draw()

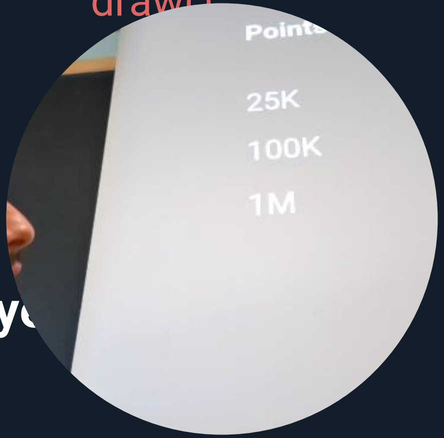
Layer (icon) draw()

Layer (polygon) draw()

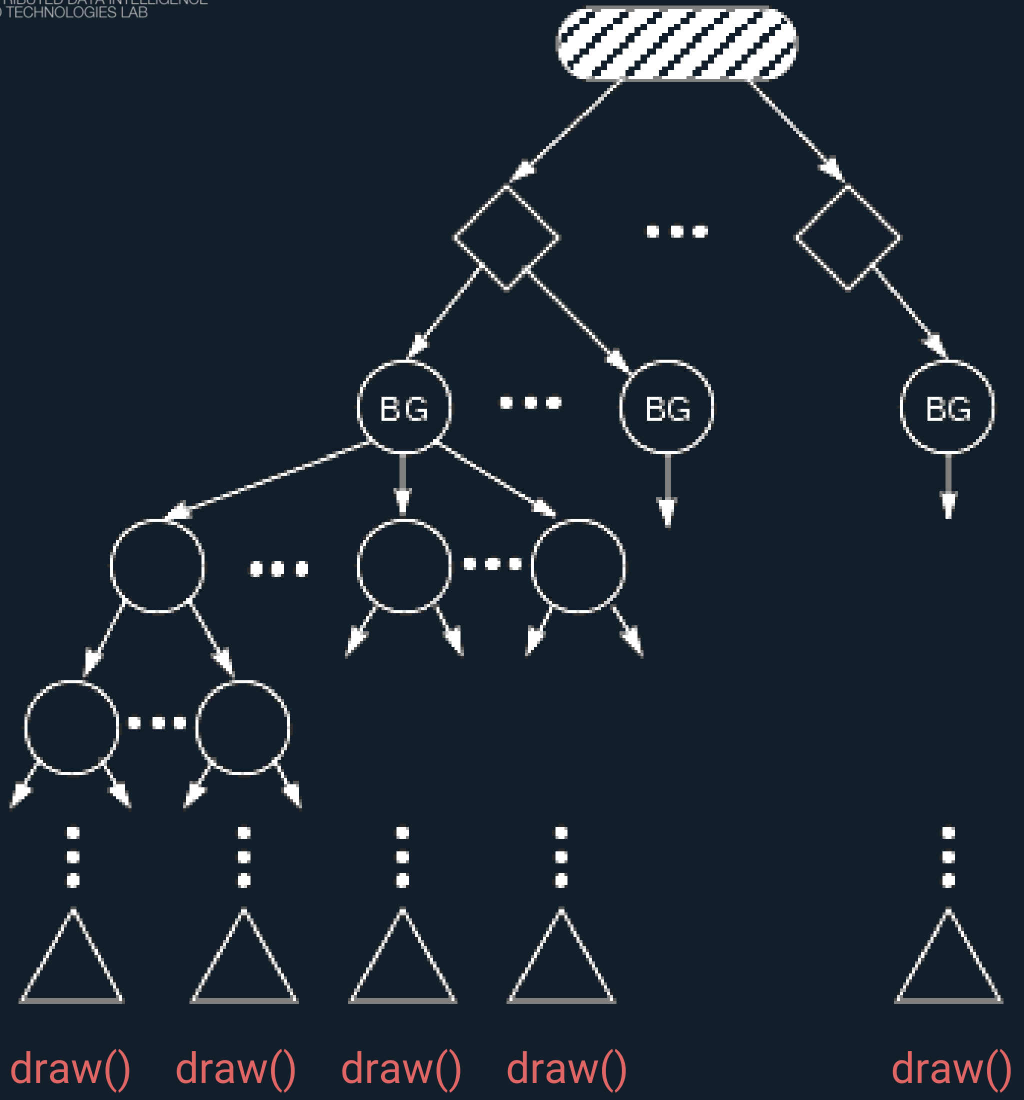
Layer (scatterplot) draw()



Laye



Layers


BG
BG
BG
draw()
draw()
draw()
draw()
draw()


Layer (path)
Layer (icon)
Layer (polygon)
Layer (scatterplot)

Layers

ScatterplotLayer

ArcLayer

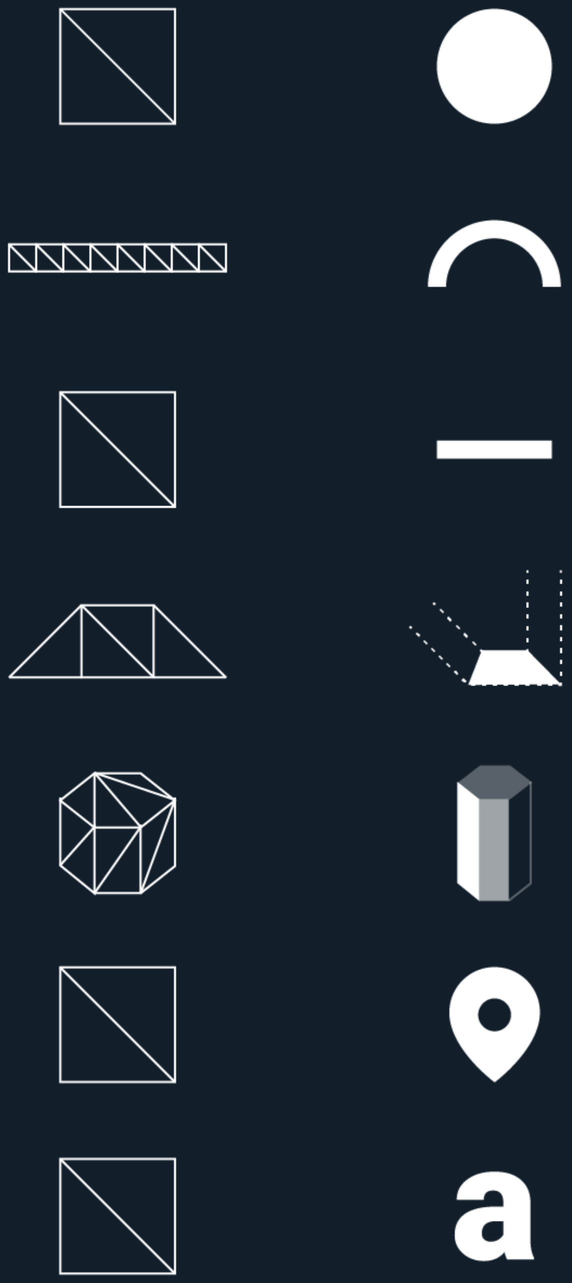
LineLayer

PathLayer

ColumnLayer

IconLayer

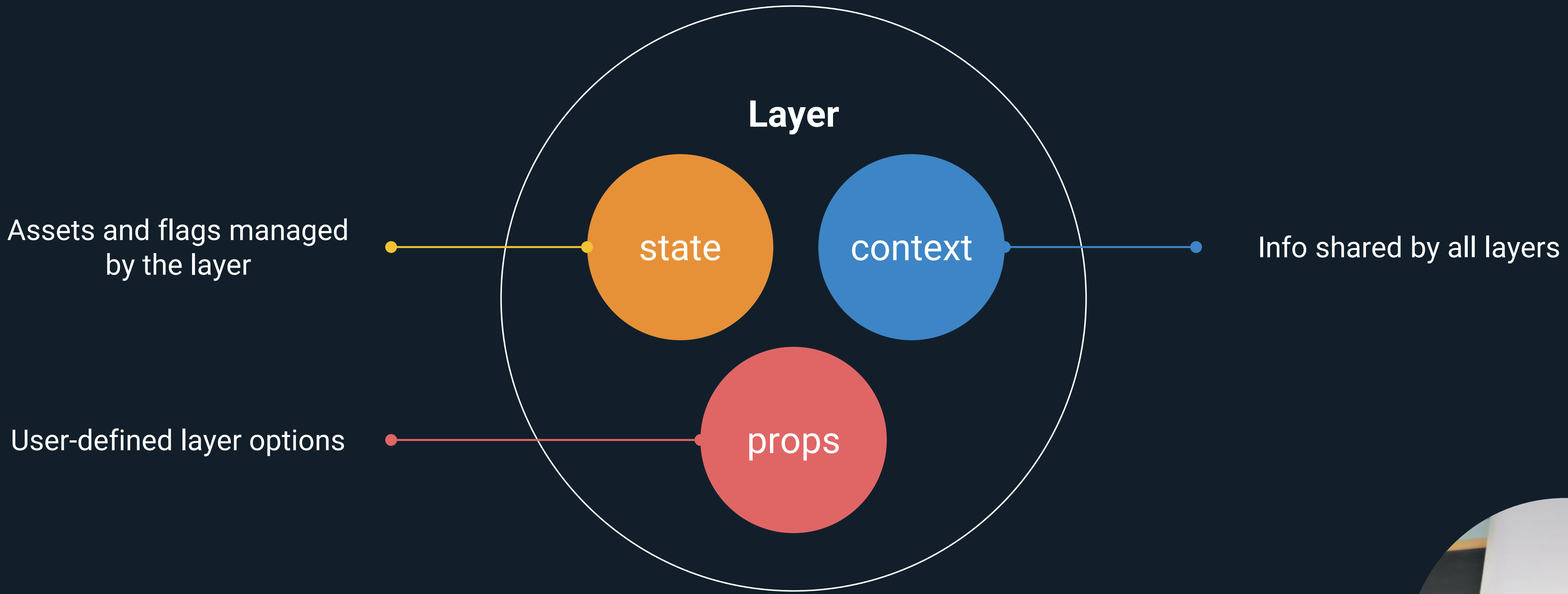
TextLayer



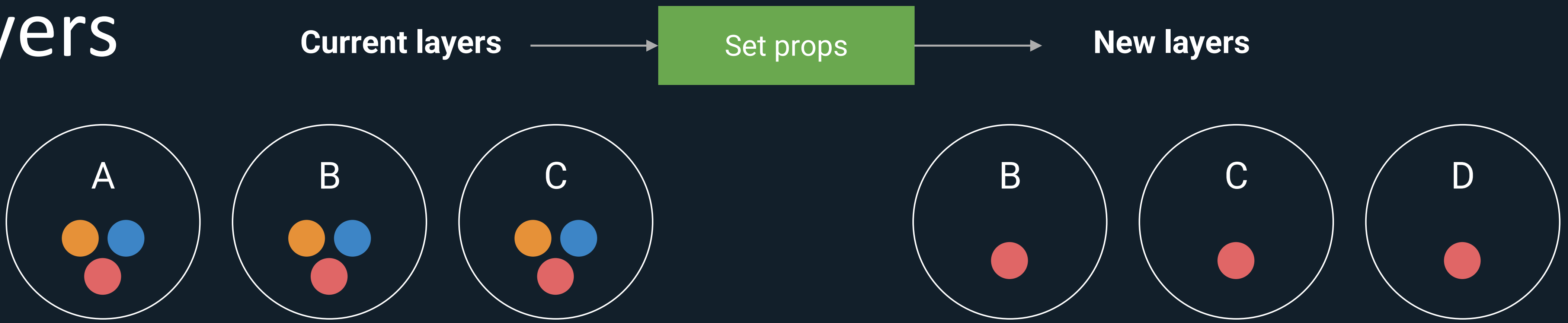
How it works

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB DISTRIBUTED DATA INTELLIGENCE AND TECHNOLOGIES LAB
SNAP4CITY

Layers

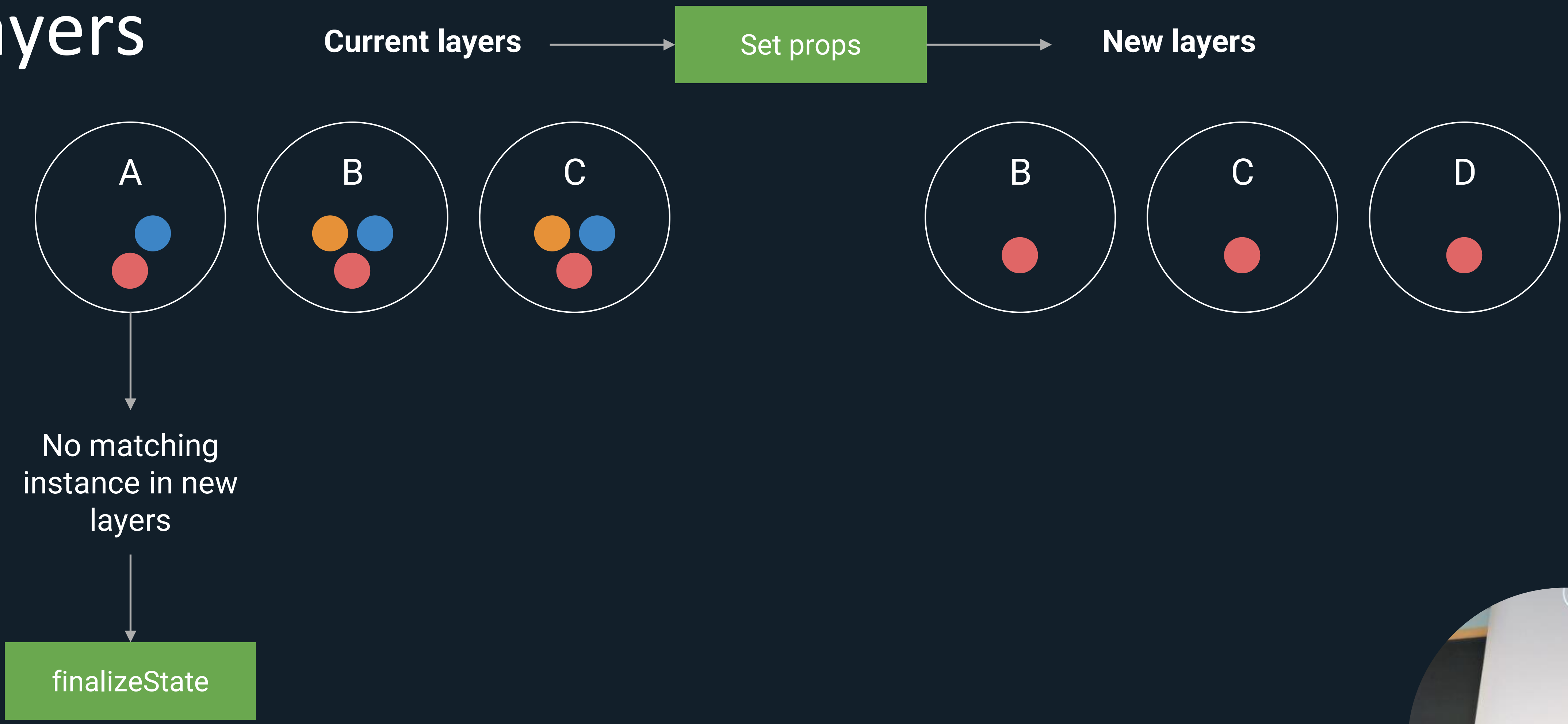


Layers

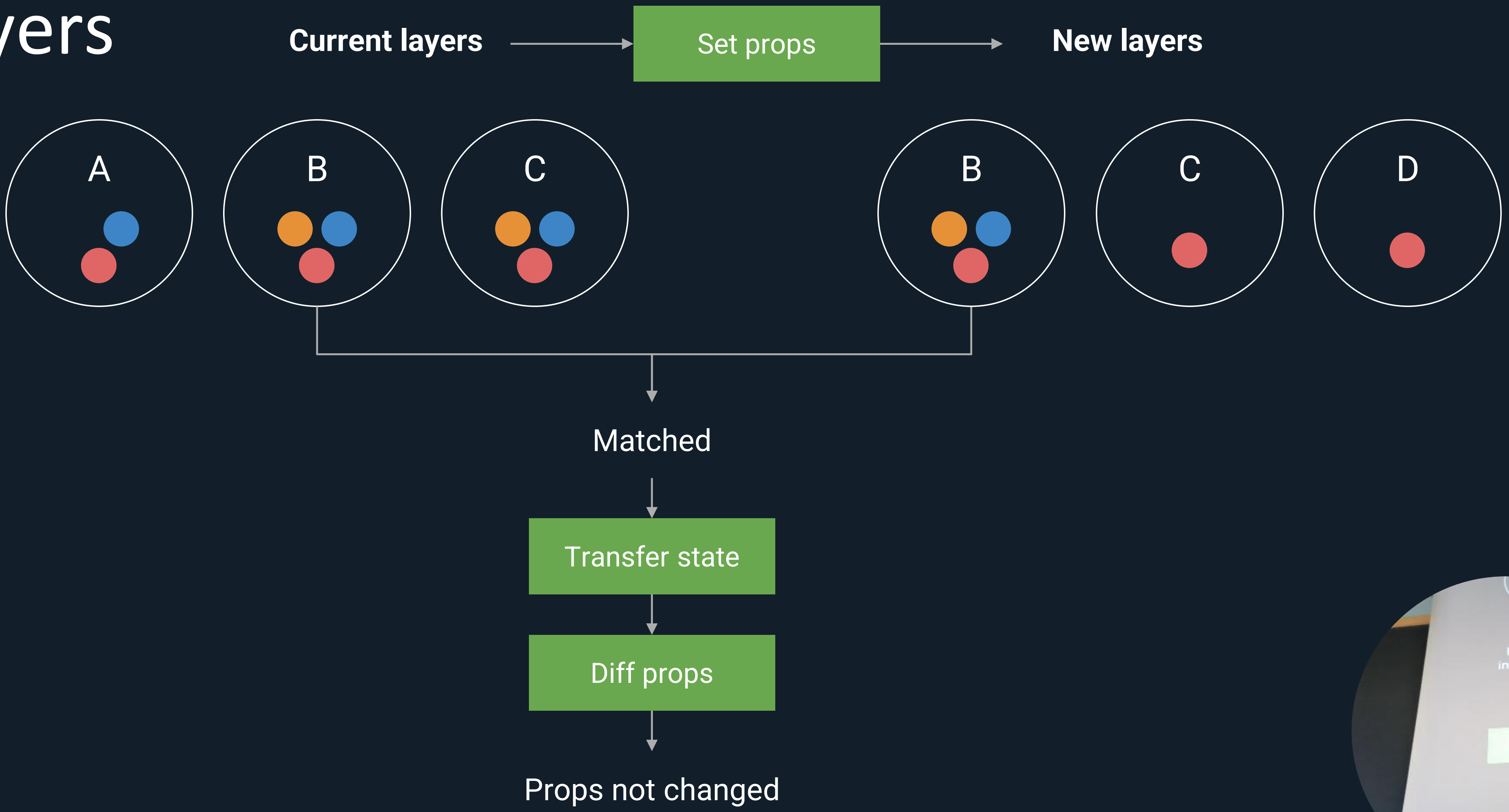

Current layers
Set props
New layers
A
B
C
B
C
D


Assets and flags
by the layer

Layers

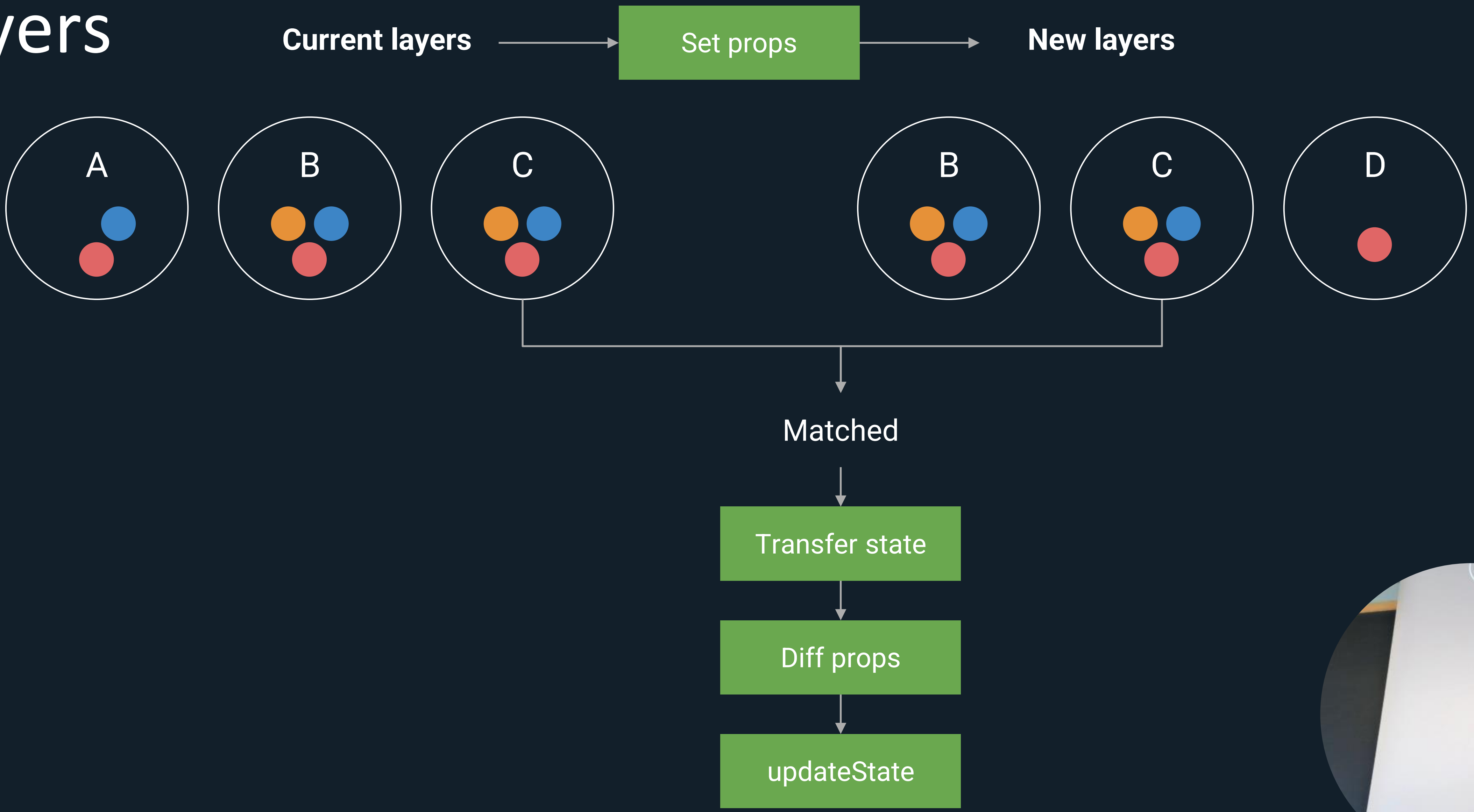

Current layers
Set props
New layers
A
B
C
B
C
D
No matching instance in new layers
finalizeState

Layers


Current layers
Set props
New layers
A
B
C
B
C
D
Matched
Transfer state
Diff props
Props not changed


No matching instance in new layers

Layers


Current layers
Set props
New layers
A
B
C
B
C
D
Matched
Transfer state
Diff props
updateState

Layers


Current layers
Set props
New layers
A
B
C
B
C
D
No matching instance in current layers
initi

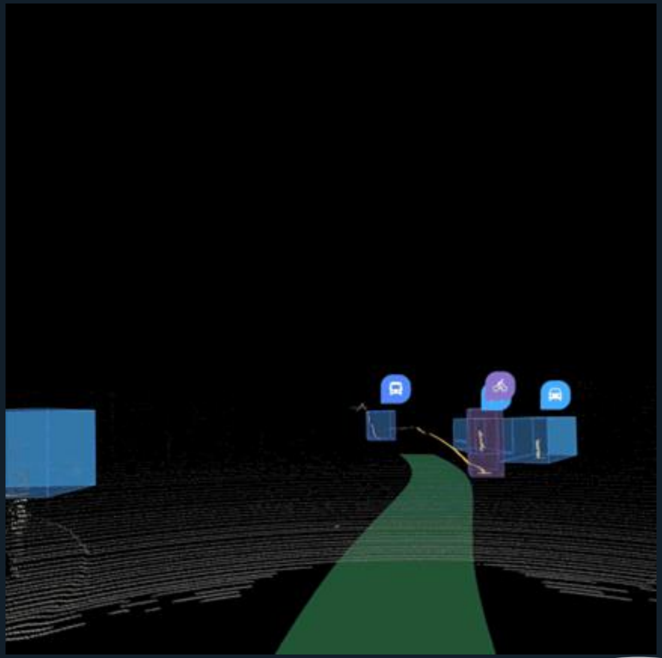
Layers Lifecycle

Lifecycle	Typical Operation	Perf Impact
initializeState	Create models, buffers, and textures	High
updateState	Pack attributes; update textures from props	Maybe high
draw	Draw to the WebGL context	Low
finalizeState	Release all WebGL resources	Low

Views and Controllers

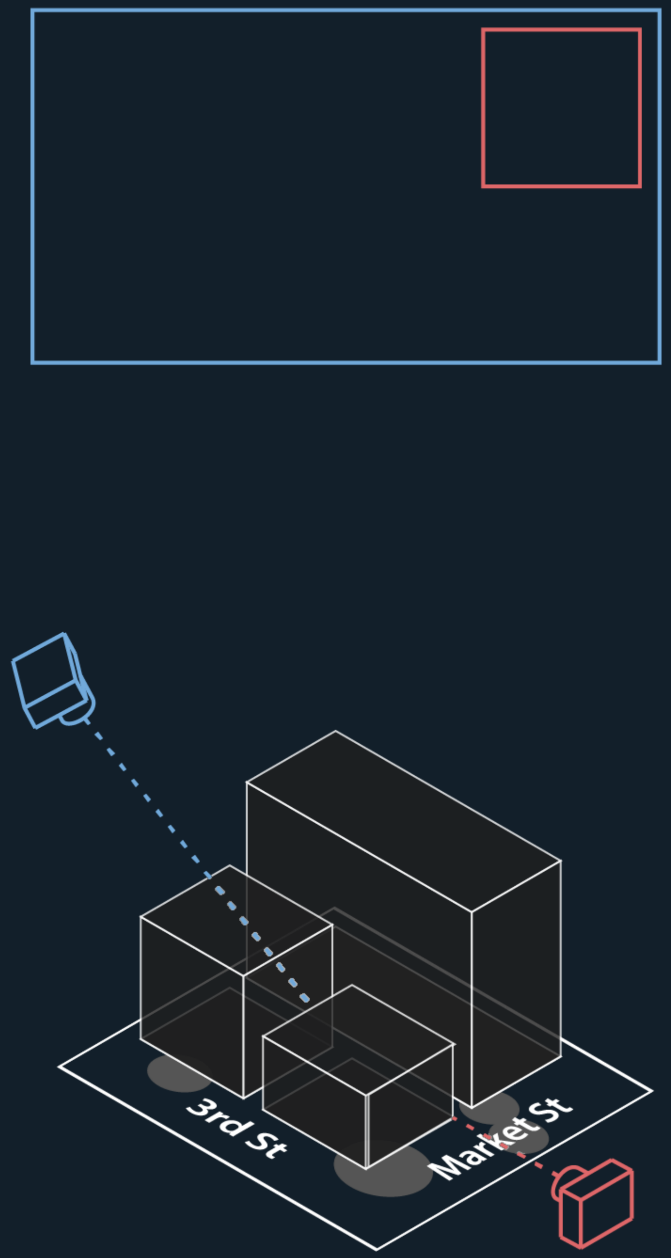
OrthographicView

Multi-View (minimap)

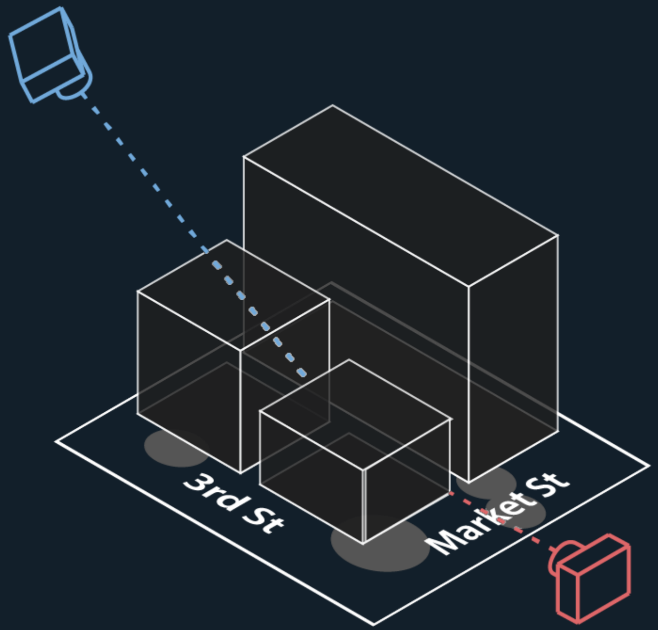


FirstPersonVie

Views



Views



View

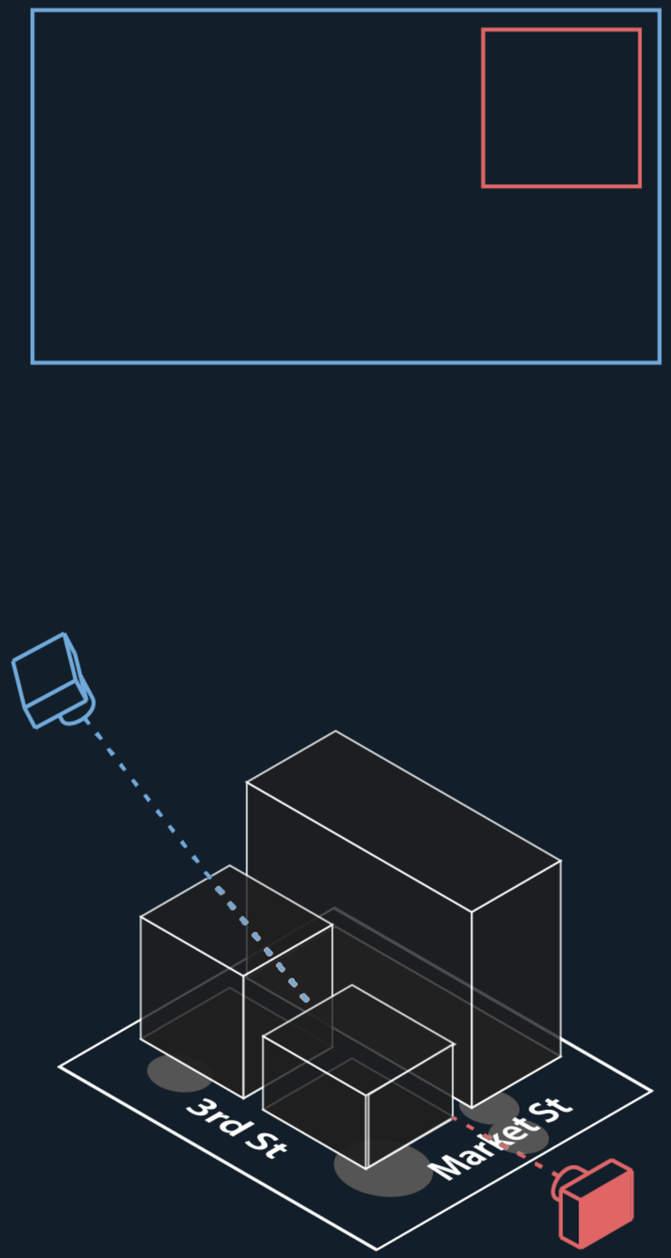
Layout Options

x
y
width
height

Projection Options

near
far
up
field of view

Views

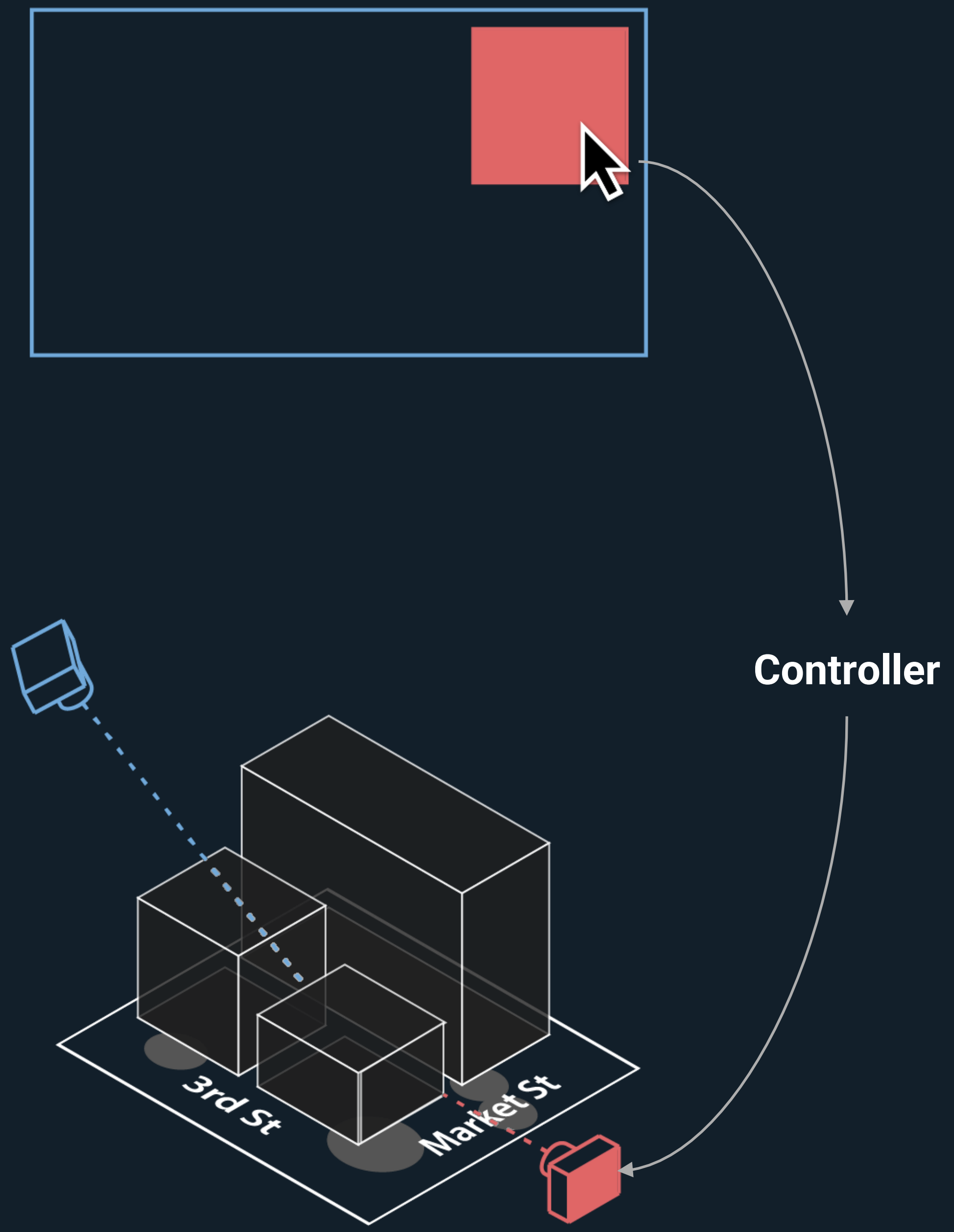


View State

Camera Position

target
zoom
pitch
bearing

Views



Picking

Other than simple visualizations most of the time a GIS web application must also offer interactive functionalities. An example is the possibility to get the characteristics of a particular building or element that the user picks on the interface. In videogames this is carried out with raytracing, that is however an expensive operation. Deck.gl implements a cheap alternative solution to do it.

The steps to achieve this are:

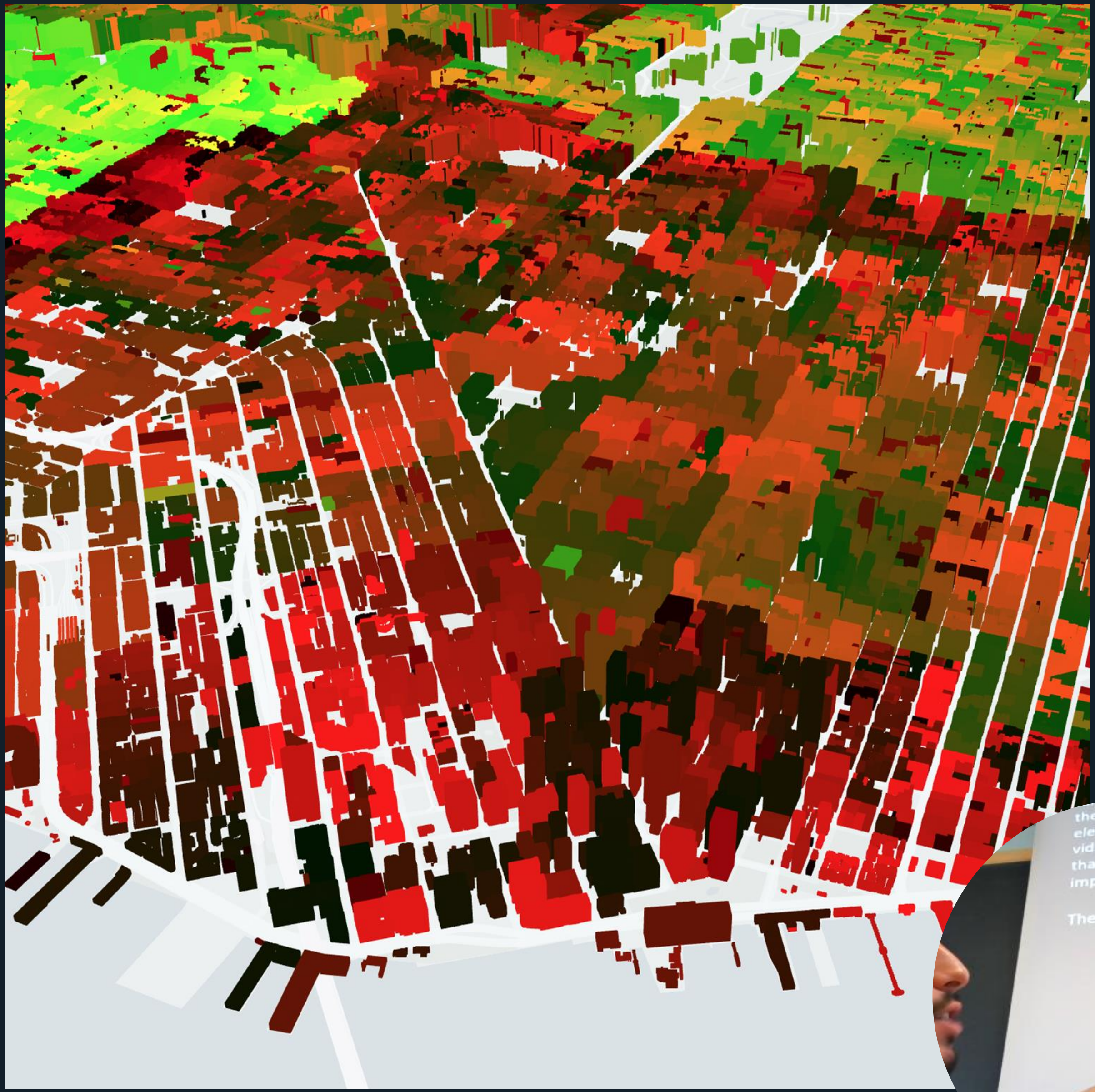
Renderbuffer

Picking

Other than simple visualizations most of the time a GIS web application must also offer interactive functionalities. An example is the possibility to get the characteristics of a particular building or element that the user picks on the interface. In videogames this is carried out with raytracing, that is however an expensive operation. Deck.gl implements a cheap alternative solution to do it.

The steps to achieve this are:

- Rendering the buildings in an offscreen canvas and color them with a value that represent the index.



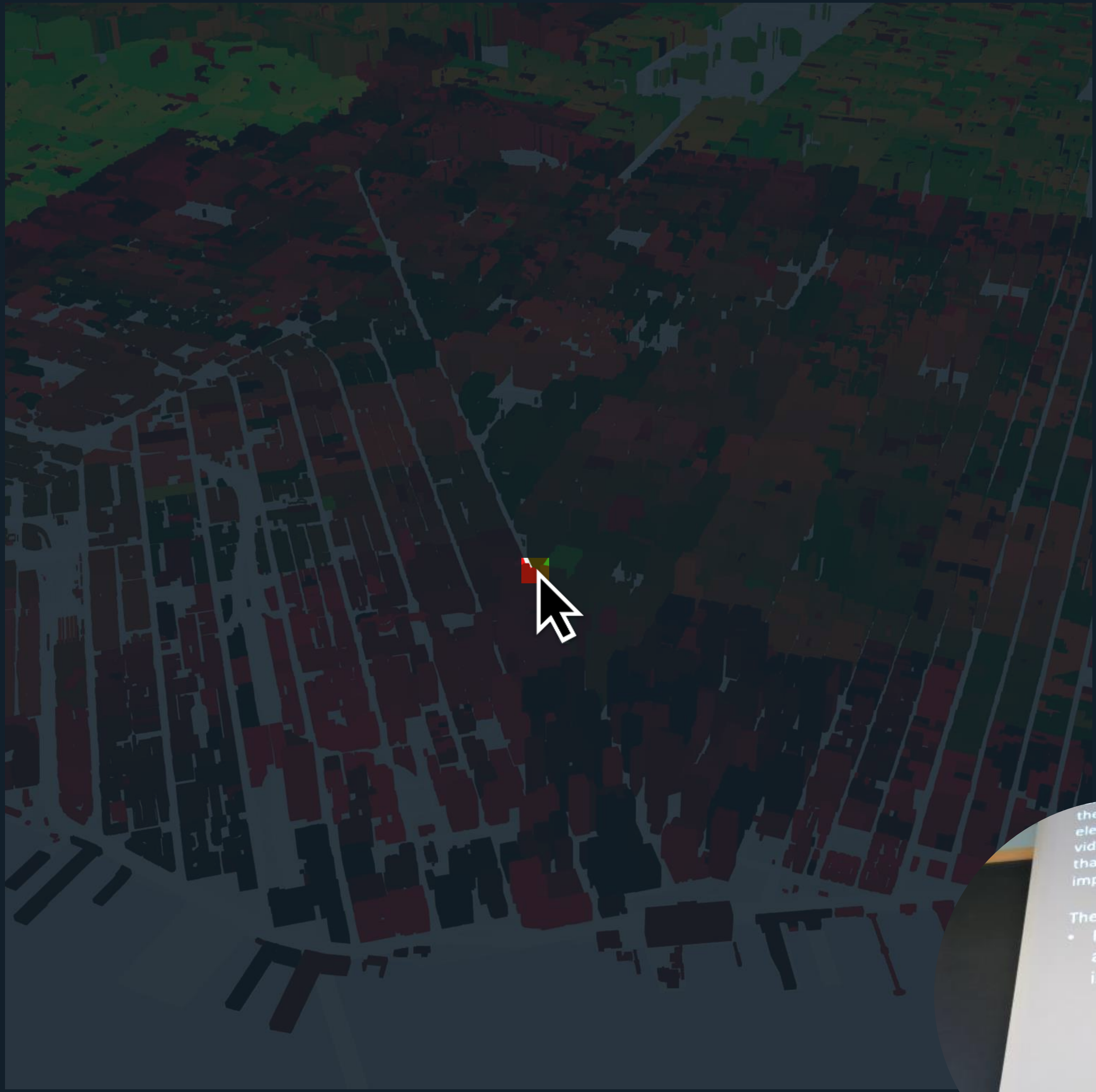
Picking Framebuffer: index-encoded

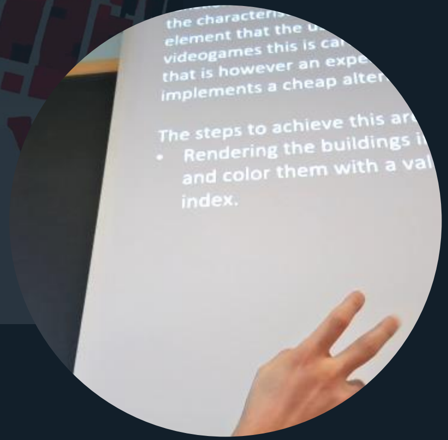
Picking

Other than simple visualizations most of the time a GIS web application must also offer interactive functionalities. An example is the possibility to get the characteristics of a particular building or element that the user picks on the interface. In videogames this is carried out with raytracing, that is however an expensive operation. Deck.gl implements a cheap alternative solution to do it.

The steps to achieve this are:

- Rendering the buildings in an offscreen canvas and color them with a value that represent the index.
- Retrieve the area of the pointer from the buffer.





Picking Framebuffer: index-encoded

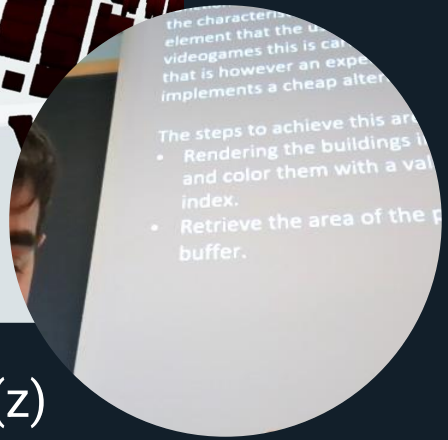
Picking

Other than simple visualizations most of the time a GIS web application must also offer interactive functionalities. An example is the possibility to get the characteristics of a particular building or element that the user picks on the interface. In videogames this is carried out with raytracing, that is however an expensive operation. Deck.gl implements a cheap alternative solution to do it.

The steps to achieve this are:

- Rendering the buildings in an offscreen canvas and color them with a value that represent the index.
- Retrieve the area of the pointer from the buffer.
- If there are multiple colors in the buffer re-color them according to the altitude Z, in order to pick the closest one.

Picking Framebuffer: attribute-incoded (z)

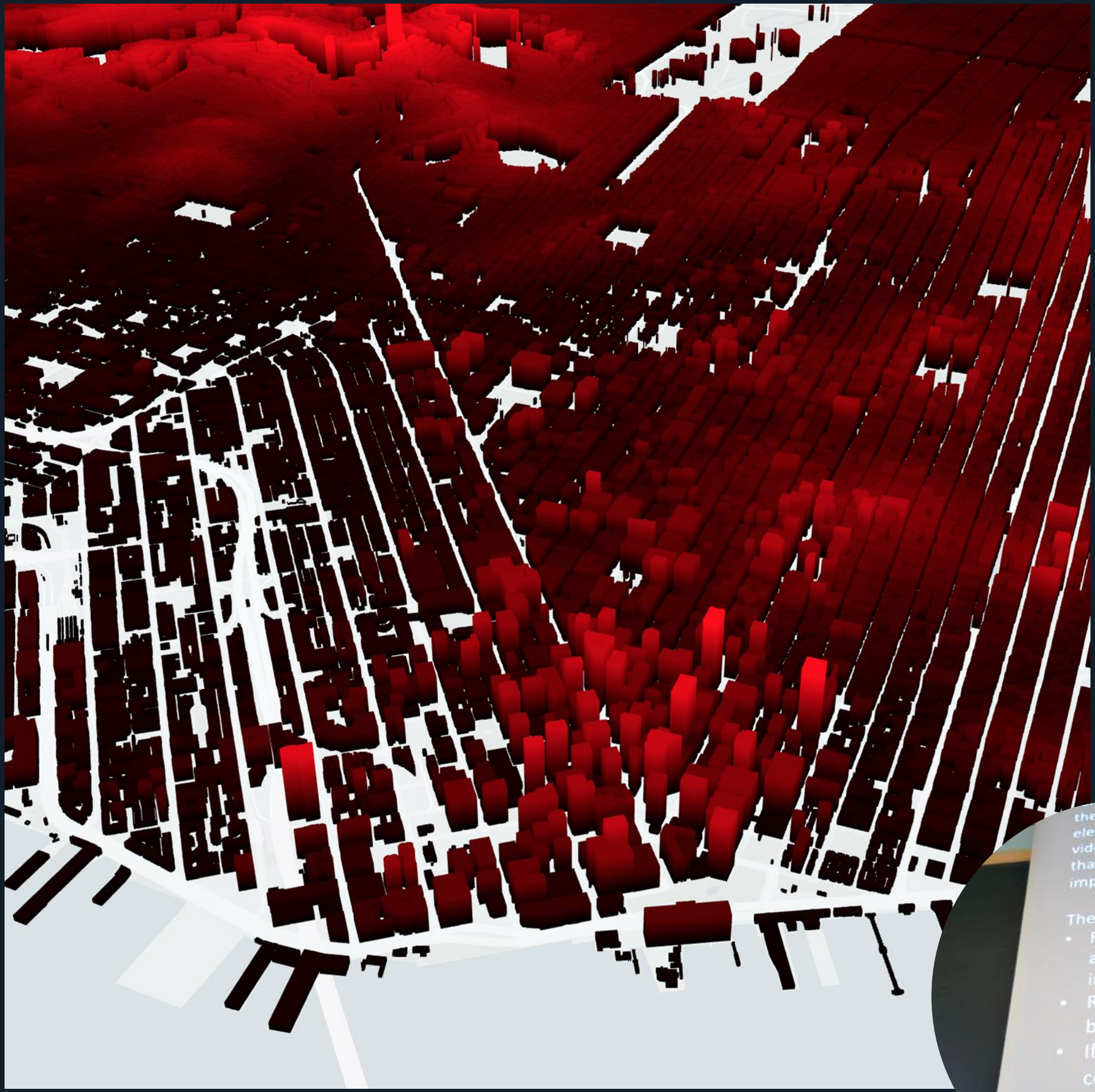


Picking

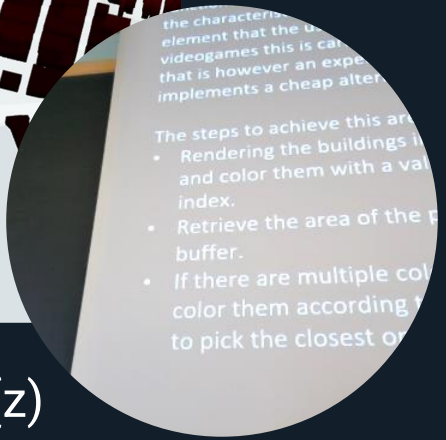
Other than simple visualizations most of the time a GIS web application must also offer interactive functionalities. An example is the possibility to get the characteristics of a particular building or element that the user picks on the interface. In videogames this is carried out with raytracing, that is however an expensive operation. Deck.gl implements a cheap alternative solution to do it.

The steps to achieve this are:

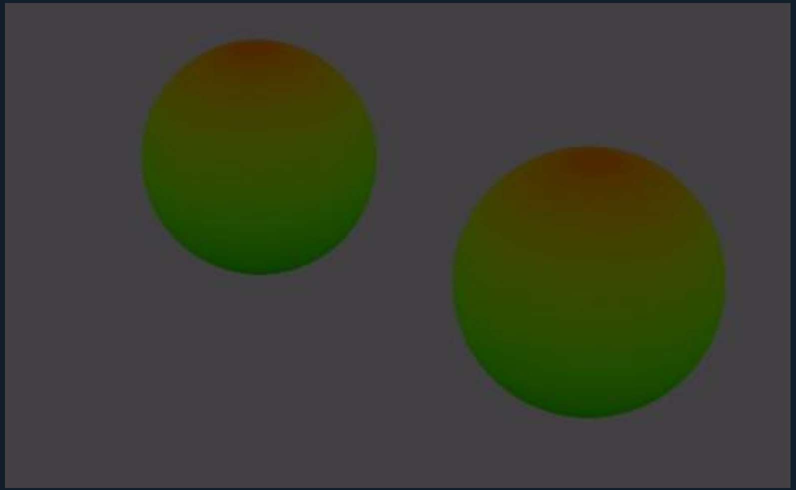
- Rendering the buildings in an offscreen canvas and color them with a value that represent the index.
- Retrieve the area of the pointer from the buffer.
- If there are multiple colors in the buffer re-color them according to the altitude Z, in order to pick the closest one.
- Return the index of the object.



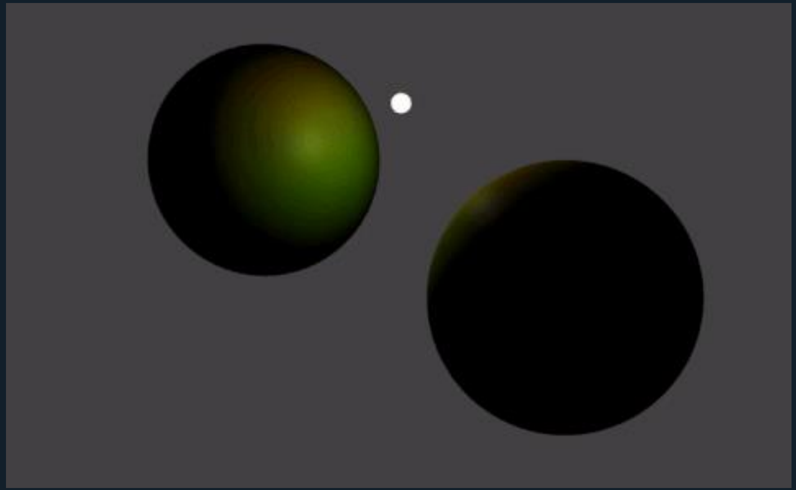
Picking Framebuffer: attribute-incoded (z)



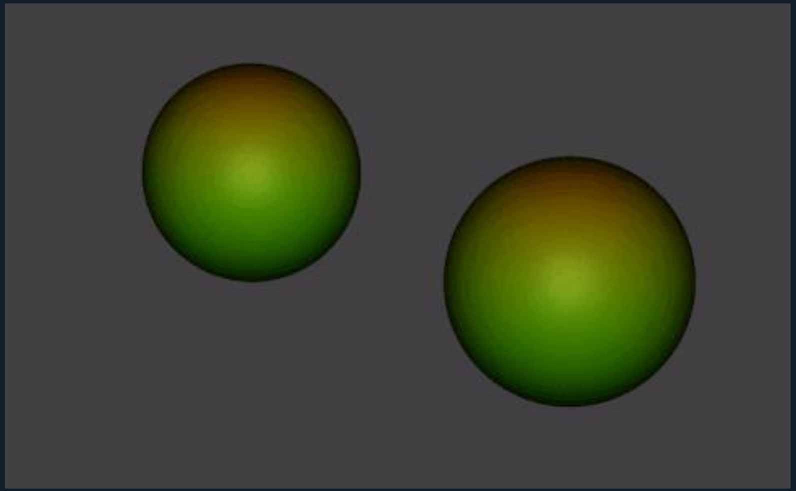
Lights



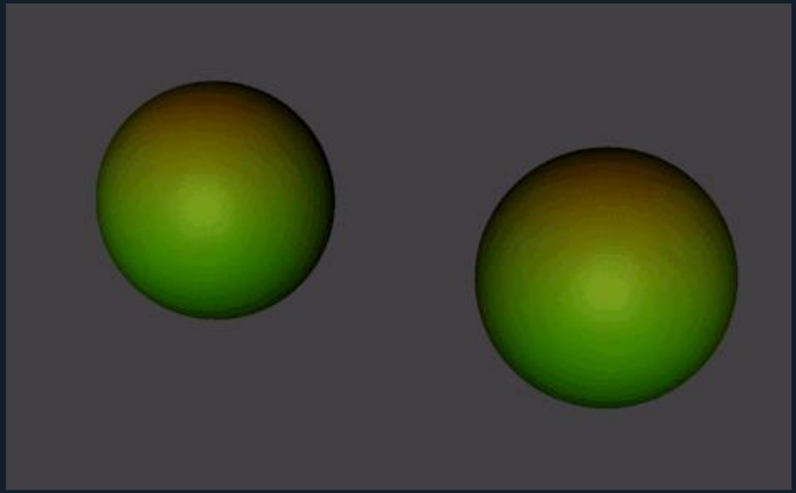
AmbientLight



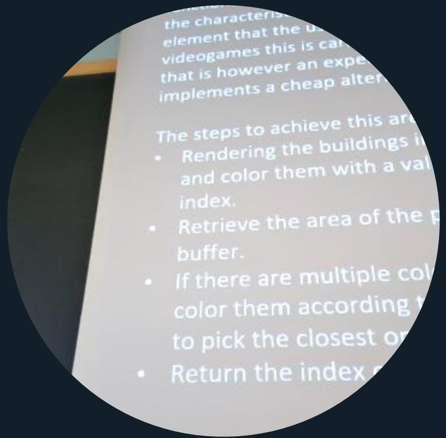
PointLight



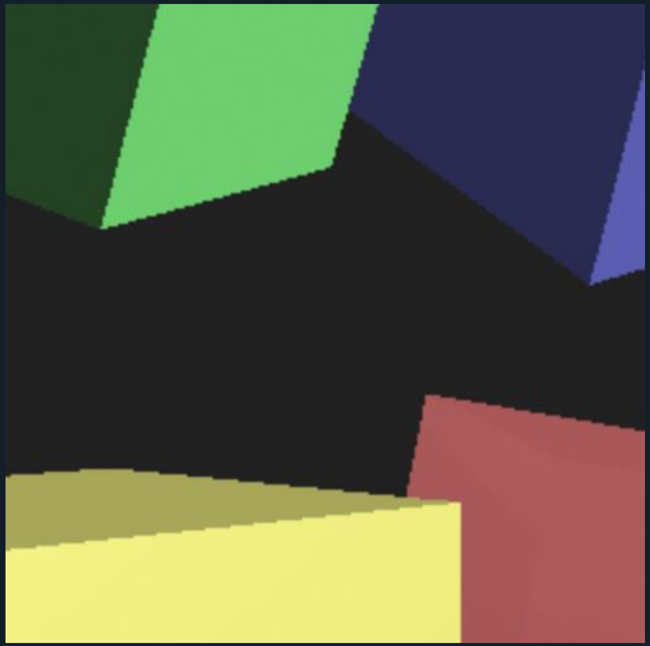
CameraLight



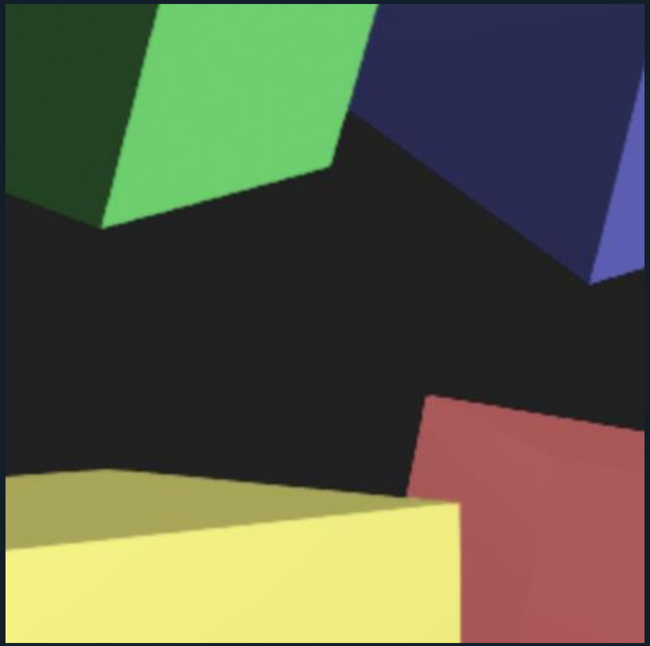
DirectionalLight



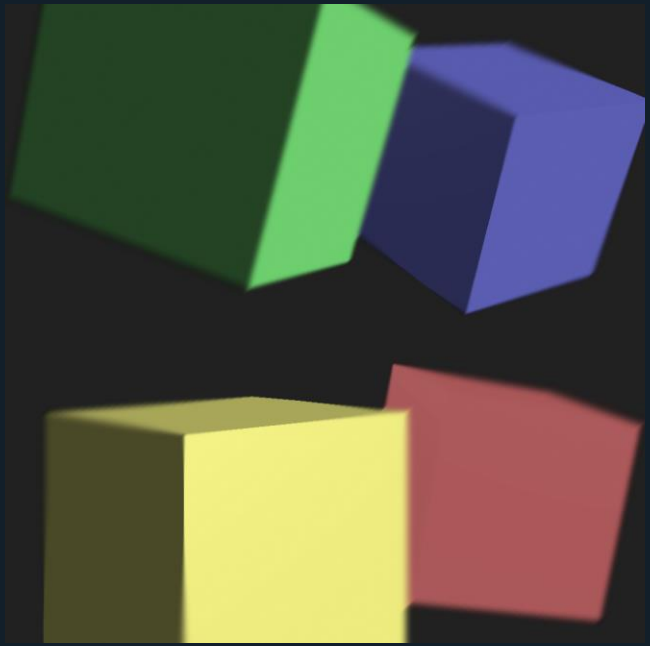
Post-Process effects



No effect

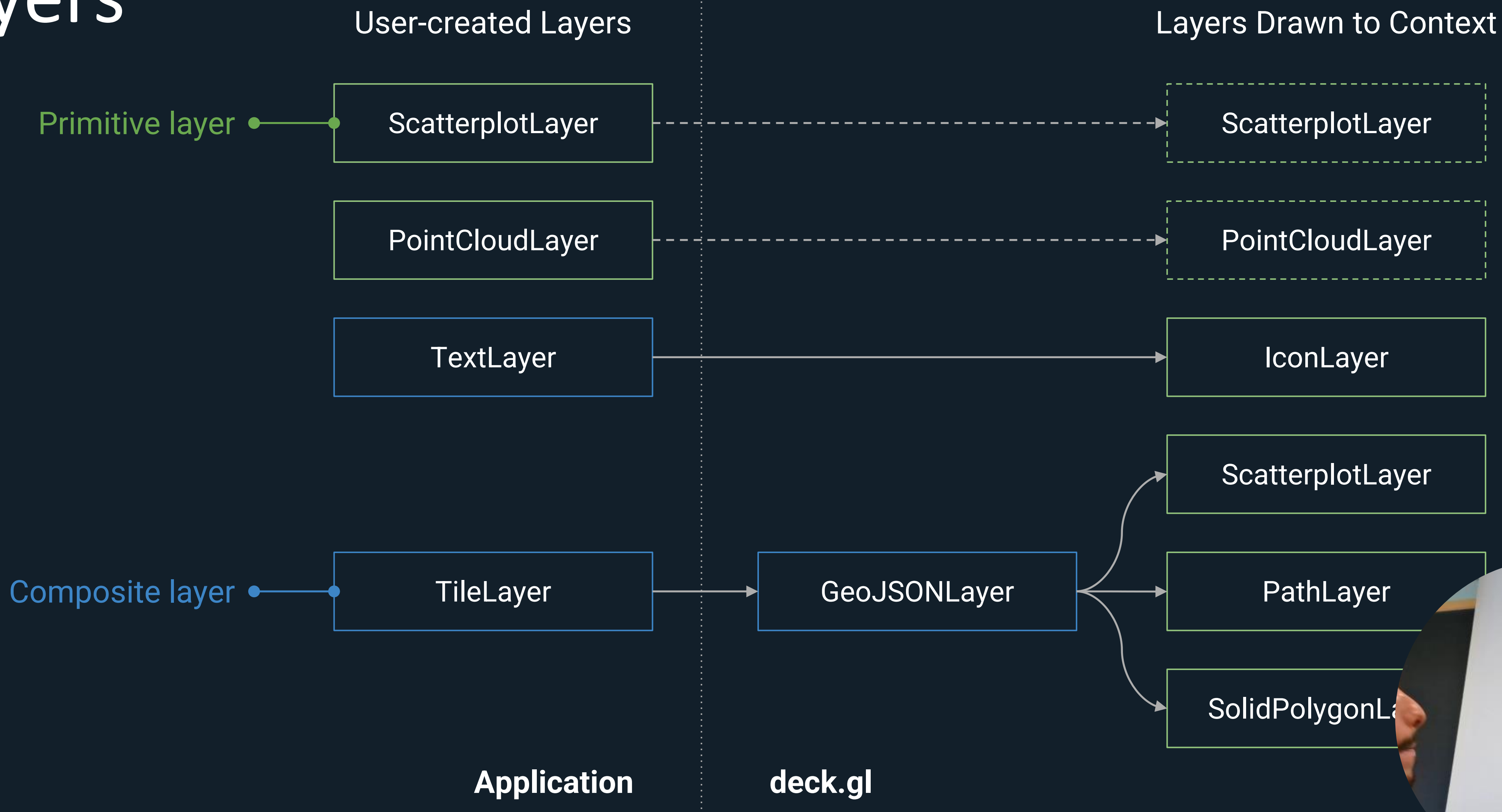


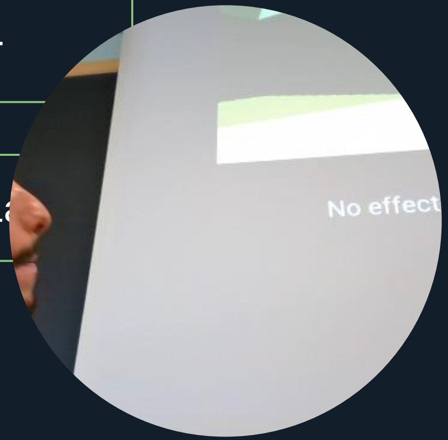
fxaa



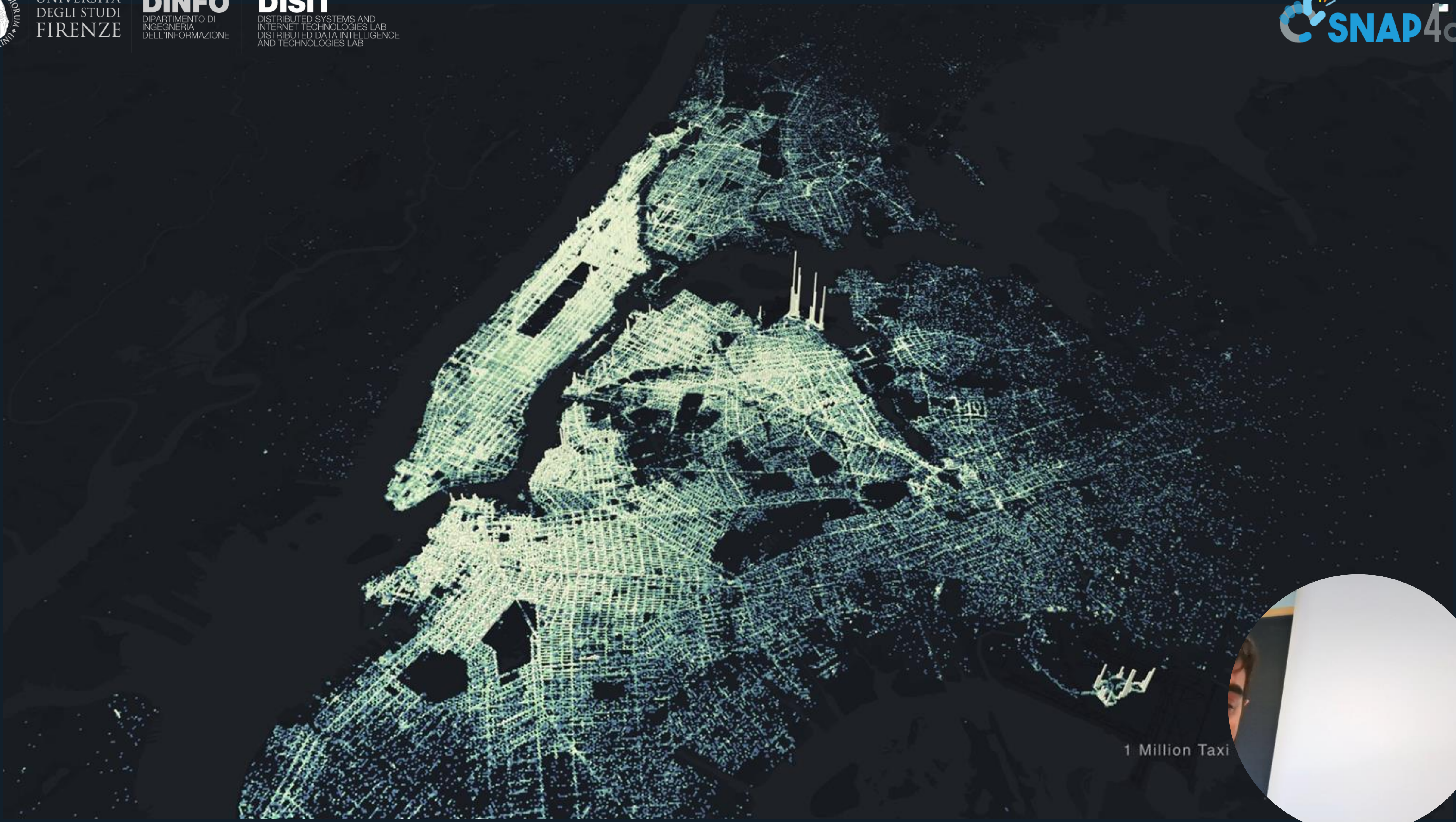
tiltShift

Layers


User-created Layers
Layers Drawn to Context
Primitive layer
ScatterplotLayer
ScatterplotLayer
PointCloudLayer
PointCloudLayer
TextLayer
IconLayer
ScatterplotLayer
Composite layer
TileLayer
GeoJSONLayer
PathLayer
SolidPolygonL
Application
deck.gl


No effect

Use Cases


1 Million Taxi

Geospatial analytics with kepler.gl

Self-driving car visualization with streetscape.gl

Implementation

Map Initialization

For the initialization of the map with Deck.gl we simply need to provide the initial canvas element or a div element that will contain it.

Then we can provide many optional parameters, some of them are:

- ViewState or InitalViewState: is an object with the initial state of the view, like longitude and latitude.
- Views and Controller: we can set the view/s of the application and their relative controllers.
- Layers: The initial layers that will be shown at the start of the application.

After the initialization we can manage all the deck application with the created object.

```
map = new deck.Deck({
    viewState: {
        longitude: -122.45,
        latitude: 37.78,
        zoom: 12,
        pitch: 30,
        bearing: 0,
    },
    canvas: document.getElementById('cvs'),
    onViewStateChange: ({ viewState }) => {
        currentViewState = viewState
        map.setProps({ viewState: { ...currentViewState } });
    },
    controller: true,
    layers: [
        // LAYERS
    ],
    getTooltip: ({ layer, object }) => {
        // TOOLTIP LOGIC
    }
});
```

Tile Layer

The TileLayer is used to render a huge number of data, subdivided in tiles. This layer is extremely useful in order to have a scalable way to render only the data that are visible to the user.

The data can also come from multiple sources and then for every tile the data will be represented in one or more layer.

For example, to render the orhtomap, we use this layer together with the BitmapLayer that is specialized to visualize an image in a certain region.

The format of the tile is a de-facto standard defined by OpenStreetMap for retrieving GIS data from the servers.

```
function createOrthomapLayer(props) {
    return new deck.TileLayer({
        id: 'OrthomapLayer',
        data: 'https://URL_TO_TILE_SERVER',

        minZoom: 0,
        maxZoom: 19,
        tileSize: 256,

        renderSubLayers: props => {
            const {
                bbox: { west, south, east, north }
            } = props.tile;

            return new deck.BitmapLayer(props, {
                data: null,
                image: props.data,
                bounds: [west, south, east,
            });
        },
        ...props,
    });
}
```

Arc Layer

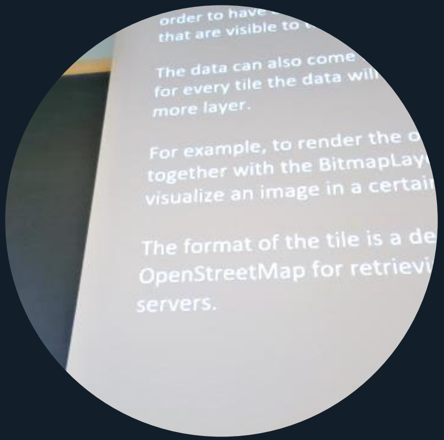
The Arc Layer is specialized in drawing arcs between two points. You just need to define the source and target positions and it will draw a curved 3D line between those points. It is possible also to define the color that can change from the source to the target, and it will appear as a gradient. Like every other layer there are a lot of different properties that you can define to customize it.

This layer is useful for example to show origin destination matrices and vehicle/people flows and trajectories.

```
function createArcLayer(props) {
    return new deck.ArcLayer({
        id: 'ArcLayer',
        data: 'https://URL_TO_DATA',

        getSourceColor: d => [Math.sqrt(d.inbound), 140, 0],
        getSourcePosition: d => d.from.coordinates,
        getTargetColor: d => [Math.sqrt(d.outbound), 140, 0],
        getTargetPosition: d => d.to.coordinates,
        getWidth: 12,
        pickable: true,

        ...props
    });
}
```



Heatmap Layer

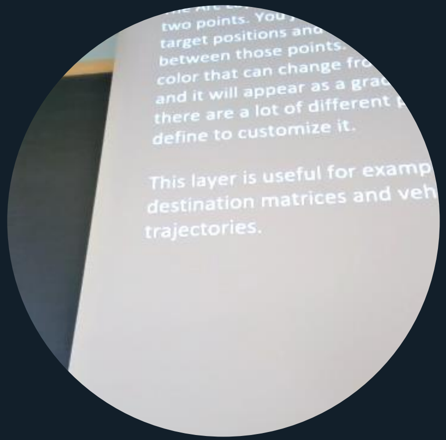
The Heatmap Layer is capable to aggregate multiple points and draw them as heatmap (a different solution with respect to use PNG images). The aggregation process is carried out in the GPU and can work with millions of points in a few seconds.

Moreover, the aggregation is redone every time the zoom changes so to let the user visualize the data in multiple scales and have a rougher or finer representation according to the actual zoom.

```
function createHeatmapLayer(props) {
    return new deck.HeatmapLayer({
        id: 'HeatmapLayer',
        data: 'https://URL_TO_DATA',

        getPosition: d => d.COORDINATES,
        radiusPixels: 25,

        ...props
    });
}
```



Hexagon Layer

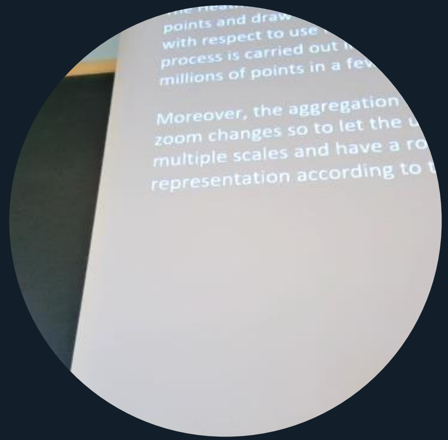
The Hexagon Layer is used to create columns that can represent different kind of information (similar to the 3D column used in the digital twin). This layer is capable to aggregate the data in GPU and show 3D pillars with a height depending of the quantity of the data that fall inside the considered area.

An example this can be used to quantify the traffic in a certain area or to visualize a particular value measured by IoT sensors.

```
function createHexagonLayer(props) {
    return new deck.HexagonLayer({
        id: 'HexagonLayer',
        data: 'https://URL_TO_DATA',

        elevationScale: 4,
        extruded: true,
        getPosition: d => d.COORDINATES,
        radius: 200,
        pickable: true,

        ...props
    });
}
```



Terrain Layer

The Terrain Layer is used to visualize the ground level with a 3D representation. In order to achieve this, it requires a raster image describing the elevation data, and a decoder function to transform the pixel RGB values in accurate elevation in meters. Textures (e.g., cadastral or satellite maps) can be superimposed on the 3D mesh.

The mesh construction is carried out in real-time using data that can be retrieved from a Geoserver using the WMS protocol. Multiple resolutions depending on the actual zoom can be used.

The algorithm to create the mesh is called Martini Tessellation, a particularly fast solution.

```
function createTerrainLayer(props) {
    return new deck.TerrainLayer({
        id: 'TerrainLayer',

        bounds: [-122.5233, 37.6493, -122.3566, 37.8159],
        elevationData: 'https://url_to_elevation',
        elevationDecoder: {
            rScaler: 2,
            gScaler: 0,
            bScaler: 0,
            offset: 0
        },
        material: {
            diffuse: 1
        },
        texture: 'https://url_to_texture',

        ...props
    });
}
```

References

- https://deck.gl/
- https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames
- https://wiki.openstreetmap.org/wiki/Raster_tile_providers
- https://github.com/mapbox/martini
- https://www.cs.ubc.ca/~will/papers/rtin.pdf