

# OpenLayers 3 workshop

4. Nyílt forráskódú térinformatikai munkaértekezlet, Budapest  
Farkas Gábor, 2015. 11. 27.

1. feladat: Készítsük fel a szerverünket! A tömörített állományban a feladateleírason kívül találhatsz egy workshop nevezetű mappát. Ezt a mappát tömörítsd ki a web szervered mappájába. Ez a mappa szervertől, és verziótól függ, az OSGEO Live CD-s telepítésnél ez a mappa a `/var/www/http`. Ökölszabályként a következő analógiát használd: általában a `/var/www` tartalmazza a szerver kiszolgált tartalmát. Amennyiben Apache szervert használsz, ha van `http` mappa, akkor oda másolj mindent, ha nincs, akkor a `/var/www` mappába. Ellenőrzésképpen nyisd meg a `map.html` oldalt a kedvenc böngésződből (`localhost/workshop/map.html`). Ha nem egy 404-es hibaüzenetet kapsz, hanem egy üres oldalt, minden remekül működik.

Amennyiben jogosultsági problémába ütközünk, változtassuk meg a `/var/www` mappa, és annak minden almappájának a tulajdonosát a mi felhasználónkra. Ezt OSGEO Live CD esetén a következő paranccsal tehetjük meg terminálból:

```
sudo chown -R user:user /var/www
```

Kérni fog egy jelszót az operációs rendszerünk, mivel csak a rendszergazda (root felhasználó) tehet ilyen módosítást. OSGEO Live esetén ez a jelszó szintén `user`.

2. feladat: Készítsünk egy egyszerű térképet. Ehhez hozzunk létre egy `map.js` nevű fájlt a kitömörített `workshop` mappában. A linkelésekkel a HTML fájlban nem kell törődnünk, már tartalmazza a `map.js` fájl dependenciaként.

A HTML fájl már tartalmaz egy eseményvezérelt utasítást. Azt mondtam neki, hogy ha a dokumentum teljesen betöltött, indítsa el az `init` funkciót. Mivel ez a funkció még nem létezik, első feladatként ezt írjuk meg! A továbbiakban, ha az ellenkezőjét nem írom, minden sort és utasítást a `map.js` fájlba írunk.

```
function init() {
```

```
}
```

Ebben a funkcióban hozuk létre a térképünket. Minden további utasítást a két kapcsos zárójel közé írunk! Következő lépésként, hozzunk létre egy szimpla térképet. Ehhez az `ol.Map` konstruktort hívjuk. Paraméterként megadjuk neki a célponot, ahová a térképet kérjük, a rétegeket, és az alapértelmezett nézetet. A réteg elem paramétereként egy forrás konstruktort használunk. Rétegből kevés van, forrásból viszont annál több. Ez a réteg hierarchia lényege az OpenLayers 3 esetén. Már most érdemes megnyitni az API dokumentációt, ami a könyvtár elemeinek használatához ad leírást. A példában használt verzió API dokumentációja a következő linken érhető el:

<http://openlayers.org/en/v3.11.2/apidoc/>.

```
var map = new ol.Map({  
  target: 'map',  
  layers: [  
    new ol.layer.Tile({  
      source: new ol.source.OSM()  
    })  
  ],  
  view: new ol.View({  
    center: [0, 0],  
    zoom: 4  
  })  
});
```

```
    })
  });
```

Mit figyelhetünk meg ezen a kódon? A konstruktorok egy szimpla objektumot fogadnak el paraméterként. Ez az egyik jellegzetessége a könyvtárnak. Bár van kivétel, ez az általános szabály. Így a könyvtár kiküszöböli a nevezetes paraméterek hiányát, ami egy fájdalmas hiányosság a JavaScript-ben. A sorrend nem számít a paramétereknél, azonban azoknak kulcs-érték pároknak kell lenniük (kettőspont az elválasztójel).

Másik sajátossága a könyvtárnak a névterek használata. Ez a logikus, jól struktúrált felépítést segíti. Minden hasonszórú konstruktor, vagy funkció, egy névtérben található. Így nyilvánvaló, hogy ha rétegre van szükségünk, akkor az `ol.layer` névtérben kell kutakodnunk, míg ha forrást akarunk, akkor az `ol.source` lesz a névterünk.

Figyeljünk oda! Az OpenLayers 3 konstruktorok nincsenek rendesen szanitálva, így hívhatóak a `new` kulcsszó nélkül is, mezei funkcióként. Ez a bővíthetőség kedvéért van így. Ha sima funkcióként hívjuk, akkor viszont az adott kódrészlet nem fog lefutni, sőt mi több, sok esetben hibaüzenetet sem generál.

3. feladat: Adjunk a térképünkhöz vektoros rétegeket! Két vektoros réteget tartalmazó állományt mellékeltem a feladathoz. Az egyik egy általános GeoJSON formátumú réteg, míg a másik egy új, TopoJSON-t használó réteg. Érdekesség, hogy a TopoJSON jelenleg nem más, mint egy irtózatosan effektív tömörítési eljárás. A vektoros rétegekhez a megfelelő réteg, és forrás konstruktorokat kell használnunk. Ezek mellett itt már van formátum, és elérési út is.

```
layers: [
  new ol.layer.Tile({
    source: new ol.source.OSM()
  }),
  new ol.layer.Vector({
    source: new ol.source.Vector({
      url: './world_countries.geojson',
      format: new ol.format.GeoJSON({
        defaultDataProjection: 'EPSG:4326'
      })
    })
  }),
  new ol.layer.Vector({
    source: new ol.source.Vector({
      url: './rivers.topojson',
      format: new ol.format.TopoJSON({
        defaultDataProjection: 'EPSG:4326'
      })
    })
  })
],
```

Mint láthatjuk, egy extra paraméterről nem beszéltünk eddig. A `defaultDataProjection`, mint azt a neve is sugallja, azt mondja meg, milyen vetületi rendszerben vannak az adataink. Mivel az alapértelmezett vetületi rendszere az OpenLayers 3-nak a gömbi Mercator (EPSG:3857), meg kell mondanunk neki, hogy az adataink WGS84 rendszerben vannak (EPSG:4326). Alapértelmezett esetben a könyvtár ezt a két vetületet ismeri.

4. feladat: Adjunk stílust a rétegeinknek! Pontosabban, csak az országhatárokat tartalmazó rétegünknek. A kék egy igazán szép árnyalatát választotta alapértelmezett stílusnak a fejlesztőcsapat, ami tökéletes a folyóink ábrázolására. Ahhoz, hogy stílust adjunk egy vektoros rétegnek, egy stílus objektumot kell építenünk a réteg objektumon belül egy konstruktorral. A stílus

különböző tulajdonságokat tartalmaz, melyek szintén konstruktorokkal adhatóak meg. Jelen esetben egy vonal, és egy területábrázolást fogunk megadni.

```
new ol.layer.Vector({
  source: new ol.source.Vector({
    url: './world_countries.geojson',
    format: new ol.format.GeoJSON({
      defaultDataProjection: 'EPSG:4326'
    })
  }),
  style: new ol.style.Style({
    stroke: new ol.style.Stroke({
      color: [0, 0, 0, 1],
      width: 3
    }),
    fill: new ol.style.Fill({
      color: [0, 255, 0, .5]
    })
  })
}),
```

Mint azt láthatjuk, színeket négy elemes tömbökkel definiálhatunk. Természetesen az első három elem az RGB színcsatornákat jelképezik, míg a negyedik trükkös lehet, az az átlátszóságot. Ebből következik, hogy az első három elemet bájt típusban kell megadnunk (0-255), míg a negyediket 0-1 között, lebegőpontos számként.

5. feladat: Kezeljünk eseményeket! Az OpenLayers 3, mint általában a JavaScript könyvtárak, két módon bővíthetőek. Építkezhetünk adott esetben már meglévő osztályokra (konstruktorokra), és készíthetünk így saját elemeket. A másik lehetőség, hogy eseményeket használunk, és eseményvezérelt funkciókat építünk ki. Gondolj ezekre úgy, mint hosszabítókra. A böngészőből már amúgy is számtalan ilyen hosszabító lóg ki, az OpenLayers 3-al épített objektumaink még többet raknak hozzá. Nekünk annyi dolgunk van, hogy megaláljuk a megfelelőt, amibe be tudjuk dugni a funkcionkat.

Ebben a feladatban annyit tegyünk, hogy írassuk ki, mit kap egy ilyen esemény során az eseményre regisztrált figyelő. Annyi a dolgunk, hogy regisztrálunk egy figyelőt a térképünk “kattintás” eseményére. Ezt a térkép objektum megépítése után, de még az `init` funkció második kapcsos zárójele előtt tegyük meg.

```
map.on('click', function (evt) {
  console.log(evt);
});
```

Ha megnyitjuk a böngészőnk konzolját (általában F12, vagy CTRL+J), és megfigyeljük a kiírt elemeket kattintásonként, bepillanthatunk egy OpenLayers 3 objektum felépítésébe. Az esemény ezt a paramétert adja tovább a figyelőnek. Tele van egy, vagy két betűs tulajdonsággal, melyek azok a paraméterek, amiket a fejlesztők elrejtettek. A használható paramétereknek (`pixel`, `coordinate`) jól olvasható, beszédes nevük van.

6. feladat: Listázzuk ki a vektoros objektumokat! Pontosabban csak azokat, amelyekre rákattintottunk. Ehhez egy, a térkép objektumból elérhető metódusra van szükségünk, mely végigiterál az objektumokon, és azokkal enged minket tovább dolgozni, melyekre rákattintottunk. Itt mutatkozik meg az OpenLayers 3 egy további sajátossága. A könyvtár konstruktorokon kívül, szinte csak metódusokat tesz elérhetővé. Így egy nagyon stabil, igazán konzisztens szerkezetet kapunk. Hátránya, hogy úgynevezett nehéz objektumokkal dolgozik, azaz egy adott objektum sokkal több memóriát foglal, mint amennyit használunk belőle.

```
map.on('click', function (evt) {
```

```

        map.forEachFeatureAtPixel(evt.pixel, function (feature) {
            console.log(feature);
        });
    });

```

Kódunk nem sokkal bővült, azonban merőben más funkcionalitással bír. Az esemény objektumából a pixel attribútumot továbbadtuk a fent említett funkciónak, mely kilistáz minden vektoros objektumot, mely az egérmutatónk alatt van egy adott kattintásnál.

7. feladat: Listázzuk ki az előbbi példában kilistázott objektumok attribútumait! Ebben a feladatban ismét egyetlen hívással bővítjük a programunkat. Ennek eredményeképpen, az kilistázza a vektoros objektumok attribútum adatait.

```

        map.forEachFeatureAtPixel(evt.pixel, function (feature) {
            console.log(feature.getProperties());
        });

```

Láthatjuk, hogy az országhatárok rétegünkhöz igen sok attribútum tartozik. Ezek közül kettő numerikus.

8. feladat: Színezzük ki az országhatár réteget attribútum adatok alapján! Ebben a feladatban az egyik numerikus oszlopot (pop\_est) fogjuk használni, mely a becsült populációt tartalmazza. Ezt úgy tudjuk megtenni, hogy a vektoros réteg stílusának nem egy objektumot határozzunk meg, hanem egy funkciót. Ez a funkció megkapja a vektoros objektumokat egyesével, valamint a jelenlegi felbontást. Ezen paraméterek alapján vár egy visszatérési értéket, mely egy tömb egy, vagy több stílus objektummal.

```

        new ol.layer.Vector({
            source: new ol.source.Vector({
                url: './world_countries.geojson',
                format: new ol.format.GeoJSON({
                    defaultDataProjection: 'EPSG:4326'
                })
            }),
            style: function (feature, resolution) {
                var pop = feature.get('pop_est');
                var color = pop > 50000000 ? [255,0,0,1] :
[0,255,0,1];

                var style = new ol.style.Style({
                    stroke: new ol.style.Stroke({
                        color: [0, 0, 0, 1],
                        width: 3
                    }),
                    fill: new ol.style.Fill({
                        color: [0, 255, 0, .5]
                    })
                });
                return [style];
            }
        }),

```

A mágikus szám a mi esetünkben az 50 000 000. Az ennél nagyobb népességű országokat pirossal, a többi zölddel jelöljük. A hagyományos if-else állítás helyett egy ternáris operátort használunk.

9. feladat: Jelenítsük meg az elkészült térképet egy 3D Földgömbön! Ehhez a feladathoz szükség lesz a map.html HTML fájl szerkesztésére. A fájlt megnyitjuk egy szövegszerkesztőben, majd a benne lévő OpenLayers 3 linkeket kicseréljük azokra, melyeket az ol3-cesium-1.10.0 mappa

tartalmaz. A végeredménynek valahogy így kell kinéznie:

```
<title>Első(?) térképem</title>
<link
href="https://cdnjs.cloudflare.com/ajax/libs/ol3/3.11.2/ol.css"
rel="stylesheet">
<link href="map.css" rel="stylesheet">
<script type="text/javascript" src="./ol3-cesium-
1.10.0/Cesium/Cesium.js"></script>
<script type="text/javascript" src="./ol3-cesium-
1.10.0/ol3cesium.js"></script>
<script type="text/javascript" src="./map.js"></script>
```

Most pedig térjünk vissza a JavaScript kódunkhoz. Bővítsük ki az `init` funkciót, és engedélyezzük benne a 3D megjelenítést. Ehhez egy konstruktort használunk az OL3-Cesium integrációs könyvtárból, majd annak az egyik metódusával engedélyezzük a 3D megjelenítést.

```
var haromD = new olcs.OLCesium({
    map: map,
    target: 'map'
});
haromD.setEnabled(true);
```

**BÓNUSZ feladat:** Bővítsük ki a 3D megjelenítést domborzattal, árnyékolással, víz textúrával, megvilágítással, és egy Nappal! Ebben a feladatban az OpenLayers 3 helyett a Cesium könyvtárhoz nyúlunk, azt bővítjük ki. Az integrációs könyvtár elérhetővé teszi a Cesium egyik alappillérét, az úgynevezett jelenetet. A jelenet tartalmaz szinte minden olyan opciót, mely a 3D megjelenítést befolyásolja, testre szabja. Mivel a jelenet objektumot könnyedén megszerezhetjük az integrációs könyvtártól, szabadon változtathatunk rajta a Cesium eszköztára segítségével.

```
var haromD = new olcs.OLCesium({
    map: map,
    target: 'map'
});
var scene = haromD.getCesiumScene();
scene.terrainProvider = new Cesium.CesiumTerrainProvider({
    url: 'http://assets.agi.com/stk-terrain/world',
    requestWaterMask: true,
    requestVertexNormals: true
});
scene.sun = new Cesium.Sun();
scene.globe.enableLighting = true;
haromD.setEnabled(true);
```

Ezzel a pár soros, igazán minimális bővítéssel egy egészen más eredményt kaptunk. A jelenet `terrainProvider` attribútumának konfigurálásával csatlakozhatuk egy Cesium által ismert domborzat formátumot közlő szerverhez. A konstruktor második paraméterével elkérjük a partokat, így a Cesium tudni fogja, hová kell víz textúrát renderelni. Figyeljük meg, hogy nagy nagyítási szinteken a könyvtár hullámszerű animációkkal is gazdagítja a végeredményt. A harmadik paraméter árnyékolással kapcsolatos információkat kér, ezzel érjük el a valóságosabb árnyékolást. A maradék két attribútum konfigurálásával létrehoztunk egy Napot, majd engedélyeztük a megvilágítást.