3.S1. Back-end

The back-end of the application is realized in the Java language and is published on a dedicated Tomcat 9.0 application server (See Fig. S1).

This takes care of both serving the html, javascript and css files to the browser and providing the services for retrieving the geometry for the layers of the Openlayers map used on the front-end. The geometry data is stored within a PostgreSQL database with PostGIS extension (see).

For the connection between the Java code and the database, the JDBC (Java DataBase Connector) driver jar file was added to the application libraries. These drivers

allow the generic Connector, PreparedStatement and ResultSet classes of Java to be used with different types of databases including Oracle, MySQL, etc. In the case of the present study, the driver named postgresql-9.4-1101.jdbc41.jar (contained in the project's WEB-INF/lib directory and linked in the application's build path) allows use with PostgreSQL databases.

The format used for the exchange of geographic data between front-end and back-end is GeoJSON, a textual format that respects the specifications of the JSON format while incorporating the geographic information in a standard form directly readable by the Openlayers library.

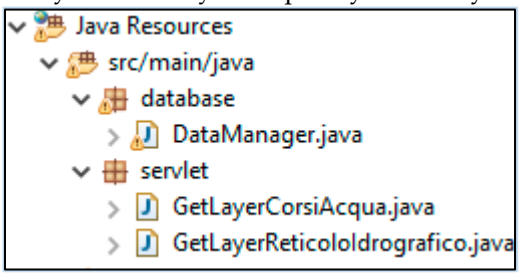


Figure S1. Project file structure - back-end

For the back-end part, the project consists of two separate Java packages. In the first, database, there is the DataManager.java class, which takes care of instantiating connections to the PostgreSQL database and is solely responsible for query execution.

In the second package, servlet, are the classes that are responsible for serving content to the front-end in response to calls from the browser. The classes within it extend the HttpServlet class and respond to the URLs indicated by the @WebServlet annotation, as in the following example: @WebServlet("/layers/getlayercorsiacqua").

3.S2. Front-end

The front-end, usable via all the latest browsers such as Google Chrome, Mozilla Firefox and Microsoft Edge, is realized in HTML, CSS and Javascript.

Some auxiliary libraries have been used for two- and three-dimensional visualization, and to facilitate access to certain types of data:

Openlayers.js: an open-source Javascript library designed to create interactive maps on the web. It provides an API that allows viewing, editing and managing geographical data from different sources, such as raster maps, vector maps and map services (OpenStreetMap, Google Maps or WMS services). It supports advanced features such as managing multiple layers, geolocalization, drawing geographical objects and interaction with GIS formats.

Three.js: for the creation of navigable three-dimensional screens, for the visualization of polygonal models. It also allows the application of textures and advanced effects, thanks to the potential of WebGL (Web Graphics Library).

Geotiff.js: for reading and managing data in geotiff format, with direct access to RAW image data.

Jquery: library of generic utilities that simplify many Javascript operations, including for example the handling of AJAX calls to the back-end.

The application is organized as a SPA (Single Page Application). This format is widely used for advanced web applications, and does not provide for the normal navigation between the various web pages as in a standard site, but manages any screen changes by means of javascript functions that regenerate the DOM, the object that takes care of maintaining a model of the current HTML document (See Fig. S2).

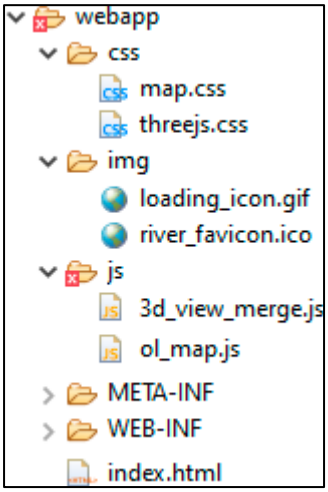


Figure S2. Project file structure - front-end

The libraries are called up within the <head> section of the index.html file. Instead of importing them into the project, it was preferred to refer to the URLs of the repositories where they are made available:

<!-- jquery --> src=https://code.jquery.com/jquery-3.4.1.min.js

<!-- Openlayers v.9.2.3 --> https://cdn.jsdelivr.net/npm/ol@v9.2.3/dist/ol.js https://cdn.jsdelivr.net/npm/ol@v9.2.3/ol.css

https://cdn.jsdelivr.net/npm/proj4@2.11.0/dist/proj4.min.js

<!-- geotiff --> https://cdn.jsdelivr.net/npm/geotiff
https://cdn.jsdelivr.net/npm/geotiff@2.0.7/dist- browser/geotiff.js
<!-- three.js v. 0.164.1 -->
https://cdn.jsdelivr.net/npm/three@v0.164.1/build/three.modu le.js
https://cdn.jsdelivr.net/npm/three@v0.164.1/examples/jsm

3.S3. Elevation assignment

The terrain elevation comes from the DEM TINITALY 1.1 and is imported into the platform thanks to the use of the WCS services offered by the portal. These allow us to download a georeferenced GeoTIFF file directly to the browser, whose raw data can be used as the basis for the creation of a three-dimensional geometry that can be viewed using Three.js. The procedure begins with a call to the WCS services, as follows:

https://tinitaly.pi.ingv.it/TINItaly_1_1/wcs?
SERVICE=WCS &VERSION=1.0.0
&REQUEST=GetCoverage &FORMAT=GeoTIFF
&COVERAGE=TINItaly_1_1:tinitaly_dem
&BBOX= [coordinate iniziali e finali della zona di interesse] &CRS=EPSG:32632
&RESPONSE_CRS=EPSG:32632 &WIDTH=500
&HEIGHT=500

Among the various parameters passed to the caller, it is possible to note that the coordinate system used is 32632, which uses metres as the unit of measurement and thus facilitates the creation of the coordinates to be inserted in the BBOX (bounding box): for an area of 5km it will be sufficient to take the central point, in this case corresponding to the coordinates of the mouse at the click, and then add and subtract 2500 to obtain the start and end points of the GeoTIFF to be downloaded. In addition to this, the required image size is also specified, 500x500 pixels, corresponding precisely to an area of 5 x 5km, which corresponds to TINITALY's definition of 10km per pixel.

The content of the response to this call is processed by the GeoTIFF.js libraries that allow the manipulation of the GeoTIFF raw data in the form of an arraybuffer, i.e. a sequence of bytes:

const arrayBuffer = event.target.response;
const tiff = await GeoTIFF.fromArrayBuffer(arrayBuffer); const image = await tiff.getImage();
const data = await image.readRasters();
renderData.demData = data[0];

In this code snippet, we create an object for the TIFF file by passing the contents of the response to GeoTIFF. From this object we extract the image, and in turn extract the raw data from it (remember that GeoTIFFs can contain multiple rasters within them, which is why the [0] element of the data array is selected, which in this case contains the data of the only image present).

This data is then saved within the renderData object, which is then passed on to the simulation.

Within the 3D visualisation, the elevation data will be realised by means of a plane with dimensions 5000 x 5000 metres, consisting of 499 vertices per side.

const geometry = new THREE.PlaneGeometry(5000, 5000, 499, 499);

The elevation values are stored within a one-dimensional array, as are the vertex positions in the Three.js plan. It will then suffice to create a loop with i ranging from 0 to the length of the array of positions, and assign the i-content of the array with the elevation data to each vertex:

position.setZ(i, data[i]);

The end result will be a mesh like the one shown in figure S3:

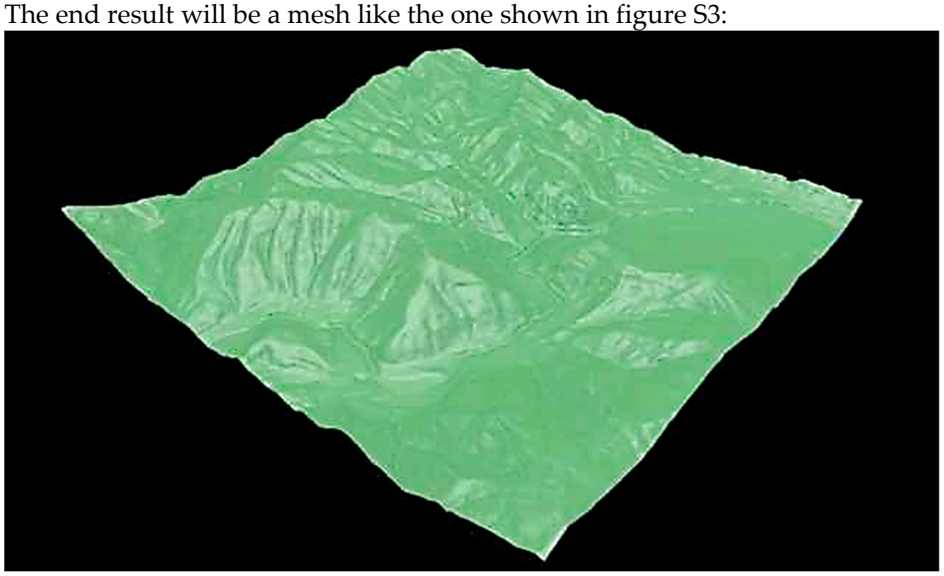


Figure S3. 3D representation of the plane with elevation.

In order to cast the simulation in as realistic an environment as possible, it is important to be able to intuitively visualize the area in which it takes place: for this purpose, having a ground texture showing satellite images of the area can make a strong contribution to the effectiveness of the tool. To achieve this, it will be necessary to have an image available, even one of arbitrary dimensions, to apply to the mesh representing the terrain.

First of all, a service was selected that could provide these images without cost, and without the need for API keys that could limit their use in the future: the choice fell on the ESRI services, which make available on their Tile Map Servers the layer called World Imagery, which contains the aerial and satellite maps of the entire planet. The service can be called up at the following URL:

https://server.arcgisonline.com/ArcGIS/rest/services/World_Imagery/MapServer/tile/{z}/{y}/{x}

Again, the OpenLayers libraries were used, generating a new map positioned outside the screen, using CSS properties:

position:fixed;left:0px;top:-1030px;width:1024px;height:1024px;z-index:20).

After creating a layer capable of accessing the ESRI service, this is added to the map, which will have an extent equal to the bounding box of the area of interest.

When the rendering of the map is complete, the following function takes care of copying the content of the image into a raster, and passing it to the 3D simulation together with the rest of the basic data:

```
const mapCanvas = document.createElement('canvas'); const size = map.getSize();
mapCanvas.width = size[0]; mapCanvas.height = size[1];
const mapContext = mapCanvas.getContext('2d'); Array.prototype.forEach.call(
map.getViewport().querySelectorAll('.ol-layer canvas, canvas.ol-layer'), function (canvas) {
if (canvas.width > 0) {
const opacity = canvas.parentNode.style.opacity || canvas.style.opacity;
mapContext.globalAlpha = opacity === '' ? 1 : Number(opacity);
let matrix;
const transform = canvas.style.transform; if (transform) {
 matrix=transform.match(/^matrix\(([^\(]*)\)$/)[1].split(',')
.map(Number);
} else {
matrix = [parseFloat(canvas.style.width) / canvas.width, 0, 0,
parseFloat(canvas.style.height) / canvas.height, 0, 0];
}
CanvasRenderingContext2D.prototype.setTransform.apply(mapContext, matrix);
const backgroundColor = canvas.parentNode.style.backgroundColor;
if (backgroundColor) { mapContext.fillStyle = backgroundColor;
mapContext.fillRect(0, 0, canvas.width, canvas.height);
}
mapContext.drawImage(canvas, 0, 0);
}},);
mapContext.globalAlpha = 1;
mapContext.setTransform(1, 0, 0, 1, 0, 0);
render3Dview(renderData.demData, renderData.alveoData, mapCanvas.toDataURL("image/png").replace("image/png", "image/octet-stream"));
```

The complete code for exporting a map as an image is available in the 'Map Export' example in the official OpenLayers documentation: https://openlayers.org/en/latest/examples/export-map.html

The data extracted in this way are passed to the 3D view creation function, and applied to the previously created plan using a standard Three.js material (See Fig. S4):

```
var materialTerreno = new THREE.MeshPhongMaterial({ color: 0x55854f, specular: 0x555555, shininess: 20, flatShading: true
});
plane = new THREE.Mesh(geometry, materialTerreno); scene.add(plane);
```



Figure S4. Texture applied to 3D plane.

Polygonal geometries from OpenStreetMap were chosen for the watercourses, imported into QGIS using the QuickOSM plug-in and subsequently saved to the PostgreSQL database serving the application. To allow their use within the 3D simulation, however, it is necessary to map these geometries onto the cells of the 5x5km area under consideration. To do this, the getFeaturesAtCoordinate() method of OpenLayers was used, which returns all features present at a certain coordinate passed in as input. Below is an extract of the function that performs this operation [20]:

```
const coordStart = ol.proj.transform([tiffData.bbox[0], tiffData.bbox[1]], 'EPSG:32632', 'EPSG:3857');
const coordEnd = ol.proj.transform([tiffData.bbox[2], tiffData.bbox[3]], 'EPSG:32632', 'EPSG:3857');
const startX = coordStart[0]; const startY = coordStart[1];
const stepX = (coordEnd[0] - coordStart[0]) / tiffData.width; const stepY = (coordEnd[1] - coordStart[1]) / tiffData.height; const arrayOut = [];
for (let x = 0; x < tiffData.width; x++) { arrayOut[x] = [];
for (let y = 0; y < tiffData.height; y++) {
let coords = [startX + (stepX * x), startY + (stepY * y)]; if
(layerCorsiAcqua.getSource().getFeaturesAtCoordinate(coords)
.length > 0) { arrayOut[x][y] = 1;
```

```
} else { arrayOut[x][y] = -1;
}}}
```

The function creates a grid of the same size as the elevation GeoTIFF, and scans this grid point by point: if a watercourse was found at that point, the corresponding element of the output array is assigned the value 1, otherwise the value -1. This array will then be inserted into the renderData object and passed to the function that is responsible for creating the 3D visualization.

Within this function, a parallelepiped geometry is created for each element of the two-dimensional array, with sides in the x- and y-direction 10m wide, and an initial height of 1m. This parallelepiped will be positioned at the height corresponding to the elevation plane for those coordinates, and the set of these parallelepipeds will constitute the initial riverbed of the watercourse on which the simulation will be performed. Below is the portion of the code that deals with generating the parallelepipeds:

```
const geometry = new THREE.BoxGeometry(10, 10, 1);
geometry.translate(x*10+offsetX,        -(y*10+offsetY), val- baseZ+2);
const colors = new Uint8Array(72); for (let r = 0; r < 24; r++) { colors[r * 3]
= 0;
colors[r * 3 + 1] = 0;
colors[r * 3 + 2] = 255;
}

const colorAttrib = new THREE.BufferAttribute(colors, 3, true);
geometry.setAttribute('color', colorAttrib); geometries.push(geometry);
```

All vertices in the geometry are assigned the colour blue (RGB 0, 0, 255). The array containing the vertex colours will have size 72, because it contains 3 integers (R, G, B) for the 24 vertices of the polygon (6 faces * 4 vertices each). When adding many polygons to the scene, a strong degradation in the performance of the application was noticed, and it was therefore decided to apply an improvement by merging all the different geometries into a single object:

```
const mergedGeometry        =
BufferGeometryUtils.mergeGeometries(geometries, false);
alveoMesh = new THREE.Mesh(mergedGeometry, materialAlveo);
scene.add(alveoMesh);
```

This solution, although the number of vertices and faces is the same, allows for a considerable improvement in performance, because the drawCalls, the calls to the webGL services responsible for visualizing the scene, are reduced, since one is performed for each different object in the scene (See Fig. S5).



Figure S5. 3D visualization of the riverbed.

3.S4. Two-dimensional map

The two-dimensional map is realised using the OpenLayers libraries. As OpenLayers does not support the 32632 (UTM32) coordinate system by default, it was necessary to import the proj4 library into the project, which is responsible for extending the projections supported by OpenLayers, via the tag:

<script src="https://cdn.jsdelivr.net/npm/proj4@2.11.0/dist/proj4.min.js"></script>

within the index.html file. At this point, using javascript, simply define the desired projection via the instruction:

```
proj4.defs("EPSG:32632", "+proj=utm +zone=32 +datum=WGS84
+units=m +no_defs +type=crs");
ol.proj.proj4.register(proj4);
```

The map will consist of three layers that are always visible:

1. the OpenStreetMap base layer
2. the watercourse layer, polygonal

3.10. the hydrographic network layer, linear

First of all, the styles with which to represent the two vector layers must be defined, as follows:

```
const styleCorsiAcqua = new ol.style.Style({ stroke: new ol.style.Stroke({
 color: "#5566ff", width: 1
}),
fill: new ol.style.Fill({ color: "#5566ff"
```

```
})});
const styleReticoloIdrografico = new ol.style.Style({ stroke: new
ol.style.Stroke({
color: "#0000ff", width: 1
})});
```

At this point, it will be necessary to indicate the source of the data for both layers: in the case of the present project, the geometries are stored on the PostgreSQL database and supplied to the front end thanks to two servlets, which execute the queries and return the result in the form of GeoJSON, a format ready to be consumed by the OpenLayers source object (the example is given for the 'Watercourses' layer, the process is equivalent for the hydrographic network):

```
const vectorSource1 = new ol.source.Vector({ loader: function (extent) {
var source = this;
$.ajax({
url: 'layers/getlayercorsiacqua', context: document.body
}).done(function (response) {
source.addFeatures((new ol.format.GeoJSON()).readFeatures(response));
});},
strategy: ol.loadingstrategy.all, projection: 'EPSG:32632'
});
```

Once the data source has been populated, it is finally possible to instantiate an object of type layer:

```
layerCorsiAcqua = new ol.layer.VectorImage({ source: vectorSource1,
name: "Corsi d'acqua", style: styleCorsiAcqua, zIndex: 5
});
```

The layer type chosen is VectorImage, which combines the flexibility and speed of vector layers with the possibility of generating cache images typical of raster vectors, for reuse in the case of tiles already previously shown on screen.

As far as the base layer is concerned, things are even simpler, since OpenLayers natively provides an object of type source.OSM that takes data directly from the OpenStreetMap services.

```
const OSMbasemap = new ol.layer.Tile({ source: new ol.source.OSM(),
});
```

In this case, the layer is of type Tile, as it shows a mosaic of pre-generated images. The map is thus complete, and we can instantiate it by inserting all the layers created previously:

```
olMap = new ol.Map({
target: 'map', layers: [
```

```
OSMbasemap, layerCorsiAcqua, layerReticoloIdrografico
],
view: new ol.View({
center: [981941, 5496717],
zoom: 9,
}),
projection: 'EPSG:32632',
});
```

Events such as clicking on a map and selecting features on certain layers can then be recorded, in order to perform the relevant operations required for the platform to function.