geojs Documentation

Release 1.0.1

Kitware, Inc.

May 25, 2021

Contents

GeoJS is a flexible library for all kinds of geospatial and 2-D visualizations from traditional point markers to 3D climatological simulations. It is designed for displaying large datasets (over 1,000,000 points) leveraging the power of WebGL. The programming interface was inspired by the widely used d3 and allows the user to generate features from arbitrary data objects with custom accessors. The API also provides custom mouse events that mimic browser level events, but work with WebGL features and even through active layers.

See the growing list of examples for a live demonstration of GeoJS's features or go to our Github repository to start hacking. If you have any questions or comments, feel free to join us on our mailing list.

CHAPTER 1

Quick start guide

1.1 Build dependencies

The following software is required to build geojs from source:

- Git
- Node.js

For testing and development, the following additional software is required:

- Python 3
- CMake

In addition, the following python modules are recommended for development and testing of geojs.

- Girder Client
- Pillow
- Requests

For testing WebGL in a headless environment, the additional packages are needed:

- mesa-utils and libosmesa6
- xvfb
- Firefox

For an example on how to install all packages for a specific OS, see *Ubuntu 14.04 Provisioning*.

1.2 Getting the source code

Get the latest geojs source code from our GitHub repository by issue this command in your terminal.

```
git clone https://github.com/OpenGeoscience/geojs.git
```

This will put all of the source code in a new directory called `geojs`.

1.3 Building the source

Inside the new `geojs` directory, you can simply run the following commands to install all dependent javascript libraries and bundle together everything that is needed.

```
npm install
npm run build
```

The compiled javascript libraries will be named `geo.min.js` and `geo.lean.min.js` in `dist/built`. The first file contains geojs including all optional dependencies. The second file is a version of geojs without any of the third party dependencies such as d3 and Hammer; if you want to use the lean bundle but need any of these dependencies, you must include them first in your page and expose them in global scope under their standard names. The bundled libraries are minified, but source maps are provided.

1.4 Using the library

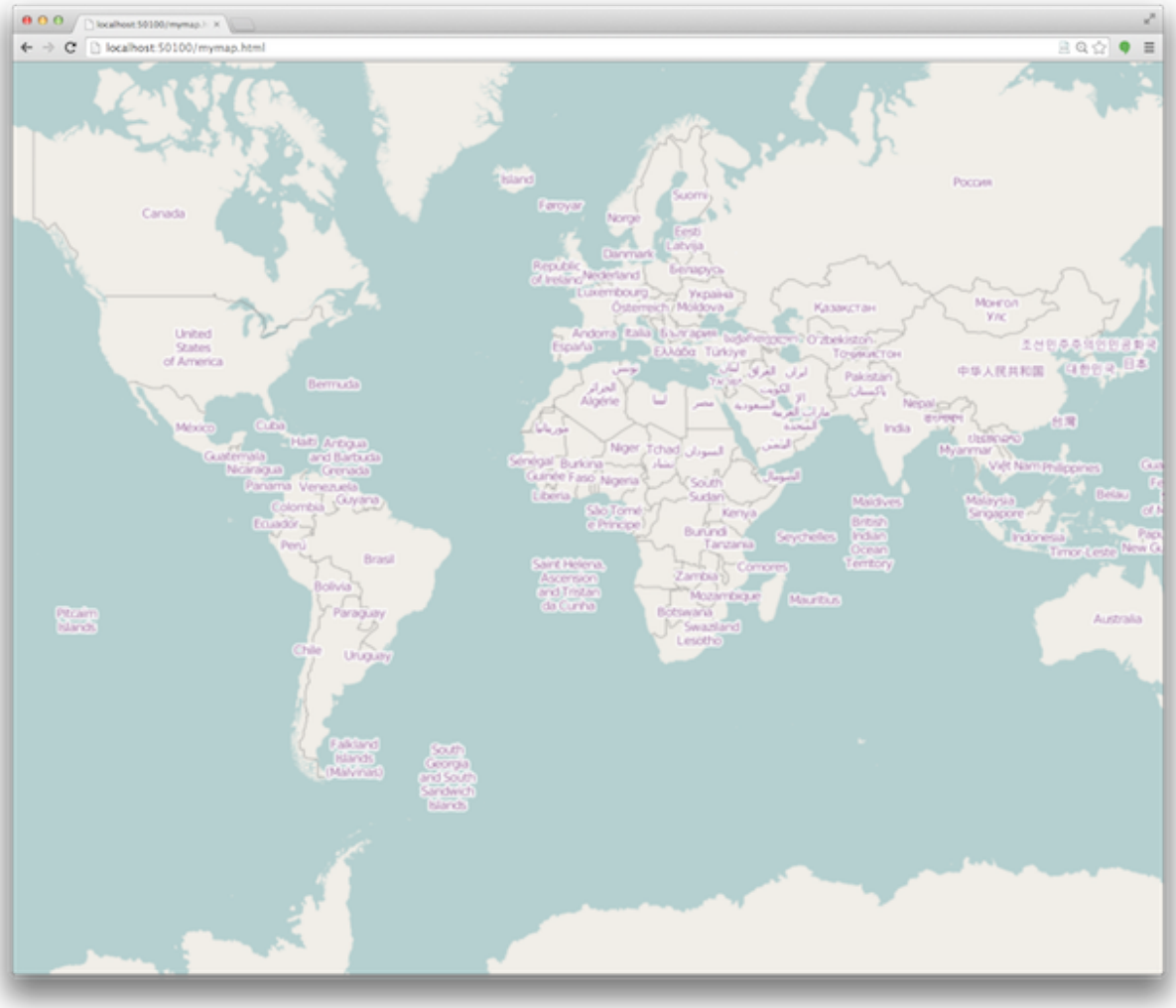
The following html gives an example of including all of the necessary files and creating a basic full map using the *osmLayer* class.

```
<head>
    <script src="/built/geo.min.js"></script>

    <style>
        html, body, #map {
            margin: 0;
            width: 100%;
            height: 100%;
            overflow: hidden;
        }
    </style>

    <script>
    $(function () {
        geo.map({'node': '#map'}).createLayer('osm');
    });
    </script>
</head>
<body>
    <div id="map"></div>
</body>
```

You can save this page into a new file at `dist/mymap.html`. To view your new creation, start up a web server with the command

```
npm run examples
```

Now, if you open up http://localhost:8082/mymap.html in your favorite webgl enabled browser, you should see a map like the following:



Additionally, you will be able to see all of the built-in examples at http://localhost:8082/examples with the example server running.

CHAPTER 2

User's guide

2.1 Dependencies

See the package.json file for a list of required and optional dependencies used by GeoJS. All libraries listed as optional dependencies are built into the main `geo.min.js` bundle, but are not present in the `geo.lean.min.js` bundle. If you want to use a subset of the optional dependencies, you can use the lean bundle and import any required dependency libraries into your page before the lean bundle, making sure that the library is exposed under its standard name in global scope.

Note: JQuery is included in the distributed bundle. Internally, this version will always be used and exposed as `geo.jQuery`. GeoJS will also set the global variable `window.$` if no other version is detected.

2.2 Software conventions

At its core, GeoJS is an object oriented framework designed to be extended and customized. The inheritance mechanism used provides an isolated closure inside the constructor to maintain private methods and variables. Prototypical inheritance is performed by a helper method called `geo.inherit`. This method copies public methods declared on the parent class's prototype. In general, classes inside GeoJS do not declare methods on the class prototype. Instead, methods are typically bound to the instance inside the constructor. This provides access to the private scope. As a consequence, a class should always call its parent's constructor before extending the implementation.

Another convention used by GeoJS eliminates the need to use the `new` keyword when constructing a new instance. This is done by checking `this` of the current context. If it is not an instance of the current class, then the constructor is called again on a new object and the result is returned to the caller.

The conventions we use result in the following boilerplate code in every class definition inside GeoJS.

```
// New class, 'B', added to the geo module derived from class, 'A'.
geo.B = function (args) {

```

(continues on next page)

(continued from previous page)

```
    // Constructors take a single object to hold options passed to each
    // constructor in the class hierarchy.  The default is usually an
    // empty object.
    args = args || {};

    // Here we handle calling the constructor again with a new object
    // when necessary.
    if (!(this instanceof geo.B)) {

        // Note: this will only happen in the constructor called by the
        // user directly, not by all the constructors in the hierarchy.
        return new geo.B(args);
    }

    // Call the parent class's constructor.
    geo.A.call(this, args);

    // Declare private variables and save overridden superclass methods.
    var m_this = this,
        s_func = this.func,
        m_var = 1;

    this.func = function () {

        // Call the super method.
        s_func();

        m_var += 1;
        return m_this;
    };

    return this;
};

// Static methods and variables can be added here.
geo.B.name = 'Class B';

// Initialize the class prototype.
geo.inherit(geo.B, geo.A);
```

Note:

- Variable naming conventions
  - The instance (`this`) is saved as `m_this`.
  - Super class methods are saved with the prefix `s_`.
  - Private variables are prefixed with `m_`.
- Methods beginning with _ are meant to be protected so they should only be called from within the class itself or by an inherited class.
- Use `m_this` to reference the instantiation inside public methods.
- Constructor options are passed inside a single object argument. Defaults should be used whenever possible.
- When possible, functions should return the class instance to support method chaining. This is particularly true for class property setters.

- In many cases, class methods return `null` to indicate an error.

2.3 Class overview

The latest version of the full API documentation is at https://opengeoscience.github.io/geojs/apidocs/geo.html.

GeoJS is made up of the following core classes. Click on the link to go to the documentation for each of the classes.

geo.map The map object is attached to a DOM element and contains all visible layers and features.

geo.renderer A renderer is responsible for drawing geometries and images on the map. This is an abstract class which serves to define the minimal interface for a renderer. Not all features are available in all renderers, and an appropriate renderer must be selected for a layer based on the features that will be used. If a renderer is requested when creating a layer, and that renderer is not supported by the current installation, a fallback renderer may be used instead and a warning sent to the console. geo.webgl.webglRenderer requires webGL support. geo.svg.svgRenderer requires the d3 library to be present.

geo.layer Layer objects are created by the map's `createLayer` method. This is an abstract class defining the interfaces required for all layers. Every layer must have a specific renderer. The following are useful layer implementations.

geo.featureLayer This is the primary container for features such as lines, points, etc.

geo.osmLayer This layer displays tiled imagery from an openstreetmaps compatible tile server.

geo.gui.uiLayer This layer contains user interface widgets that should generally be placed on top of all other layers.

geo.feature Feature objects are created by the featureLayers's `createFeature` method. Features are created from an arbitrary array of objects given by the `feature.data` method. Properties of the features can be given as constant values or as functional accessors into the provided data object. The styles provided are largely independent of the renderer used; however, some differences are necessary due to internal limitations. The following are feature types currently available.

- geo.pointFeature
- geo.lineFeature
- geo.pathFeature
- geo.graphFeature
- geo.vectorFeature

Note: Some features types are only available for specific renderers.

geo.gui.widget This is an abstract interface for creating widgets that the user can interact with.

- geo.gui.domWidget
- geo.gui.svgWidget
- geo.gui.sliderWidget
- geo.gui.legendWidget

geo.mapInteractor This class handles all mouse and keyboard events for the map. Users can customize the mouse and keyboard bindings through this class.

geo.fileReader This is an abstract class defining the interface for file readers. Currently, the only implemented reader is geo.geojsonReader, which is an extendable geojson reader.

2.4 Coordinate systems

A major component of GeoJS's core library involves managing several coordinate systems that are used to keep layers aligned on the screen. The following conventions are used in GeoJS's documentation and codebase when referring to coordinates:

Latitude/longitude coordinates Expressed in degrees relative to the WGS84 datum as objects using keys `x` for longitude and `y` for latitude. Longitudes are assumed to be in the range `[-180, 180]`. Some map projections (such as the default `EPSG:3857`) are periodic in `x` and handle automatic wrapping of longitudes.

GCS coordinates Expressed in standard units (usually meters) as defined by Proj.4, which is used to perform coordinate transformations internally. The coordinate system `EPSG:4326` is equivalent to latitude/longitude coordinates described above. Points in these coordinate systems are given as an object with keys x and y providing the horizontal (left to right) and vertical (bottom to top) positions respectively. GCS coordinates have an optional `z` value that is `0` by default. The units of `z` should be expressed in the same units as `x` and `y`.

Display coordinates Expressed in units of pixels relative to the top-left corner of the current viewport from top to bottom.

World coordinates These are the coordinates used internally as coordinates of the 3D scene in much the sense as defined in 3D graphics. The world coordinates are a rescaled and translated version of the GCS coordinates so that the world coordinates of the current viewport is near `1` in each axis. This is done to provide well conditioned transformation matrices that can be used accurately in contexts of limited precision such as WebGL or CSS. In order to achieve this, the world coordinate system is dynamic at run time and will change as the user pans and zooms the map. By convention, the world coordinates are given relative to a dynamic "scale" and "origin". Changes to these values trigger events on the map that allow layers and features to respond and update their views as necessary.

Layer coordinates To allow flexibility for layer/renderer implementation, layers are allowed to use their own custom coordinate system via the functions `toLocal` and `fromLocal`. Features inside a layer should always pass coordinates through these methods to access the coordinates inside the layer's context.

Feature coordinates Features have a GCS property attached to them that should be taken to mean a geographic coordinate system for the data passed into the feature. For features such as points, coordinates are automatically transformed into the map's GCS by Proj.4, then transformed into world coordinates, and finally into layer coordinates before being passed to the layer's rendering methods.

2.5 Coordinate transformation methods

To facilitate uniform transformation between the many coordinate systems used inside a map object, there are many available transformation methods provided in the core API. These methods vary from being useful to all users of the library to methods that are only relevant to developers interacting with low level renderers or wishing to optimize performance. The following is a list of transform methods present in the library as well as example uses for them.

`geo.map.gcsToDisplay/displayToGcs(c, gcs)` This is the most common transformation method that converts from a geographic coordinate system into pixel coordinates on the map. If no GCS is given, the method will assume the coordinate system of the map. For example, to get the lat/lon of the point under the mouse you would get the pixel coordinates relative to the map's container and pass them to this method as `c` in `map.displayToGcs(c, 'EPSG:4326')`.

geo.map.gcsToWorld/worldToGcs(c, gcs) This performs the conversion to internal world coordinates that are scaled and translated to deal with round off errors. This method is made available so that layers can use a consistent base coordinate system from which the camera transforms are derived.

geo.layer.fromLocal/toLocal(c) This converts between world space and a custom coordinates system defined by each layer. The default implementation of these methods returns the original coordinate unmodified, but layers can choose to override this behavior as needed. Users generally do not need to call this method unless they are interacting with the low level context of the layer.

geo.camera.worldToDisplay/displayToWorld(c, width, height) This converts between world space coordinates and display pixel coordinates given a viewport size. In addition to these methods, the camera class provides access to the raw transformation matrices for layers that can make use of them directly. For layers supporting CSS there is also a `camera.css` property that returns a CSS transform representing the current camera state.

CHAPTER 3

Developer's guide

Note: This guide assumes you have cloned and built the geojs repository according to the *Quick start guide*.

To run all of the tests, you will need the optional packages and python modules listed there.

Geojs employs several different frameworks for unit testing. These frameworks have been designed to make it easy for developers to add more tests as new features are added to the api.

3.1 Code quality tests

All javascript source files included in the library for deployment are checked against ESLint for uniform styling and strict for common errors patterns. The style rules for geojs are located in the `.eslintrc` file in the root of the repository. These tests are preformed automatically for every file added to the build; no additional configuration is required. You can run a quick check of the code style outside of CMake by running `npm run lint`.

3.2 Code coverage

Code coverage information is generated automatically for all headless unit tests by Karma's test runner when running `npm run test`. The coverage information is submitted to codecov and cdash after every successful Travis run.

3.3 Headless browser testing

Geojs uses PhantomJS for headless browser testing of core utilities. Unfortunately because PhantomJS does not support webgl at this time, so code paths requiring webgl must be either mocked or run in an environment such as xvfb.

The headless unit tests should be placed in the `tests/cases/` directory. All javascript files in this directory will be detected by the Karma test runner and executed automatically when you run `npm run test`. It is possible to debug

these tests in a normal browser as well. Just run `npm run start` and browse to http://localhost:9876/debug.html. The test runner will automatically rebuild the tests as you modify files so there is no need to rerun this command unless you add a new file.

There are a number of utilities present in the file `tests/test-utils.js` that developers can use to make better unit tests. For example, a mocked vgl renderer can be used to hit code paths within webgl rendered layers. There are also methods for mocking global methods like `requestAnimationFrame` to test complex, asynchronous code paths in a stable and repeatable manner. The Sinon testing library is also available to generate stubs, spies, and mocked methods. Because all tests share a global scope, they should be careful to clean up all mocking and instrumentation after running. Ideally, each test should be runnable independently and use jasmines `beforeEach` and `afterEach` methods for setup and tear down.

3.4 Headless WebGL testing

To fully test code that uses WebGL, a browser with WebGL is required. If xvfb, osmesa, and Firefox are installed, some tests can be run in a virtual frame buffer that doesn't require a display. May of these tests depend on additional data which can be downloaded by using CMake and running ctest.

For example, running

```
cmake /path/to/geojs
make
xvfb-run -s '-ac -screen 0 1280x1024x24' ctest -VV -R ffheadless
```

will run the headless WebGL tests. After the data for tests is downloaded, the tests can also be run via `npm run test-webgl`, which assumes that `xvfb-run` is available.

The headless unit tests that require WebGL should be placed in the `tests/gl-cases/` directory. When tests are run in a normal browser via `npm run start`, the webgl tests are included.

Many of these tests compare against a baseline image. If a test is changed or added, new baselines can be generated and optionally uploaded via the script built into `test/baseline_images.py`.

If a test fails, the specific test will be reported by the test runner, and the base and test images are saved in the `images` subdirectory of the build directory. The images have the base name of the test and end in `-base.png` for the reference image, `-test.png` for the current test, and `-diff.png` for a difference image where areas that are different are highlight (using resemblejs, the default highlight color is pink).

Unless an image comparison test fails, images are not automatically saved. To save all images, add the environment variable `TEST_SAVE_IMAGE=all` to the test command or set this parameter in CMake.

Note: Typically, CMake is used to build outside of the source tree. This means you would create a new directory somewhere and point cmake to the geojs source directory. You may need to rerun `cmake` and `make` after making changes to your code for everything to build correctly. Try running `ccmake /path/to/geojs` for a full list of configuration options.

Examples and tests that need to run in a standard browser should be tested by creating an entry in the `tests/headed-cases/` directory. To run these tests in a normal browser, run `npm run start` and browse to http://localhost:9876/debug.html?test=all. Since the browser's direct screen output is used, the browser must be running on the same machine as the `npm run start` command.

3.5 Release Process

To make a new GeoJS release:

- Update the version number in package.json.
- Update the CHANGELOG.md file with changes since the last release
- Commit to GitHub and merge to master
- Tag the commit on GitHub with a tag of the form vX.Y.Z (e.g., v0.17.0).
- After the release appears on GitHub, update the release notes with the changes since the last release.

Tagging a commit on the master branch will trigger a build on travis-ci.com, at the end of which the new version will be published to npm and the build artifacts will be pushed to the tagged version on GitHub.

CHAPTER 4

Provisioning for Development

4.1 Ubuntu 18.04

This shows how to set up a build and test environment in Ubuntu 18.04.

These instructions will probably work for any Ubuntu release from 18.04 onward. They assume a basic installation.

Add nodejs to the sources so it can be installed

```
wget -qO- https://deb.nodesource.com/setup_8.x | sudo bash -
```

Install required packages (you may want to also include cmake-curses-gui for convenience in configuring CMake options)

```
sudo apt-get install --yes \
    cmake \
    firefox-esr \
    git \
    imagemagick \
    libjpeg-dev \
    libpango1.0-dev \
    mesa-utils \
    nodejs \
    python-pip \
    xauth \
    xvfb
```

Checkout the GeoJS source and change to the source directory

```
git clone https://github.com/OpenGeoscience/geojs.git
cd geojs
```

Install node modules

```
npm install
```

Build GeoJS and run some basic tests

```
npm run build
npm run lint
npm run test
```

Note that some of the tests measure speed, and therefore may fail if you are running on slow hardware or in a limited virtual machine.

Use CMake to create additional tests and make to download test data

```
cmake .
make
```

Run the headless WebGL tests

```
ctest -VV -R headless
```

Run all tests

```
xvfb-run -s '-ac -screen 0 1280x1024x24' ctest --output-on-failure
```

Install python packages

```
pip install --user girder-client
```

Generate new baseline images for the WebGL tests

```
python test/baseline_images.py --xvfb --generate --upload --verbose
```

CHAPTER 5

Testing infrastructure

5.1 baseline_images

CHAPTER 6

Indices and tables

- genindex
- search