geojs Documentation

Release 0.2.0

Kitware, Inc.

March 25, 2015

Contents

GeoJS is a flexible library for all kinds of geospatial visualizations from traditional point markers to 3D climatological simulations. It is designed for displaying large datasets (over 100,000 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 mimick 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. GeoJS is in active development and many components are still being refactored in preparation for a stable release. 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

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

- Python 2.7
- Make
- CMake
- Pillow
- Requests
- Selenium

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`. Geojs depends on another repository called vgl. In order to get the vgl source code as well you will need to go into the `geojs` directory and tell git to download the vgl submodule.

```
git submodule init
git submodule update
```

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
grunt
```

Compiled javascript libraries will be named `geo.min.js` and `geo.ext.min.js` in `dist/built`. The first file contains geojs and vgl bundled together. The second file contains all of the dependent libraries. The bundled libraries are minified, but source maps are provided

1.4 Using the library

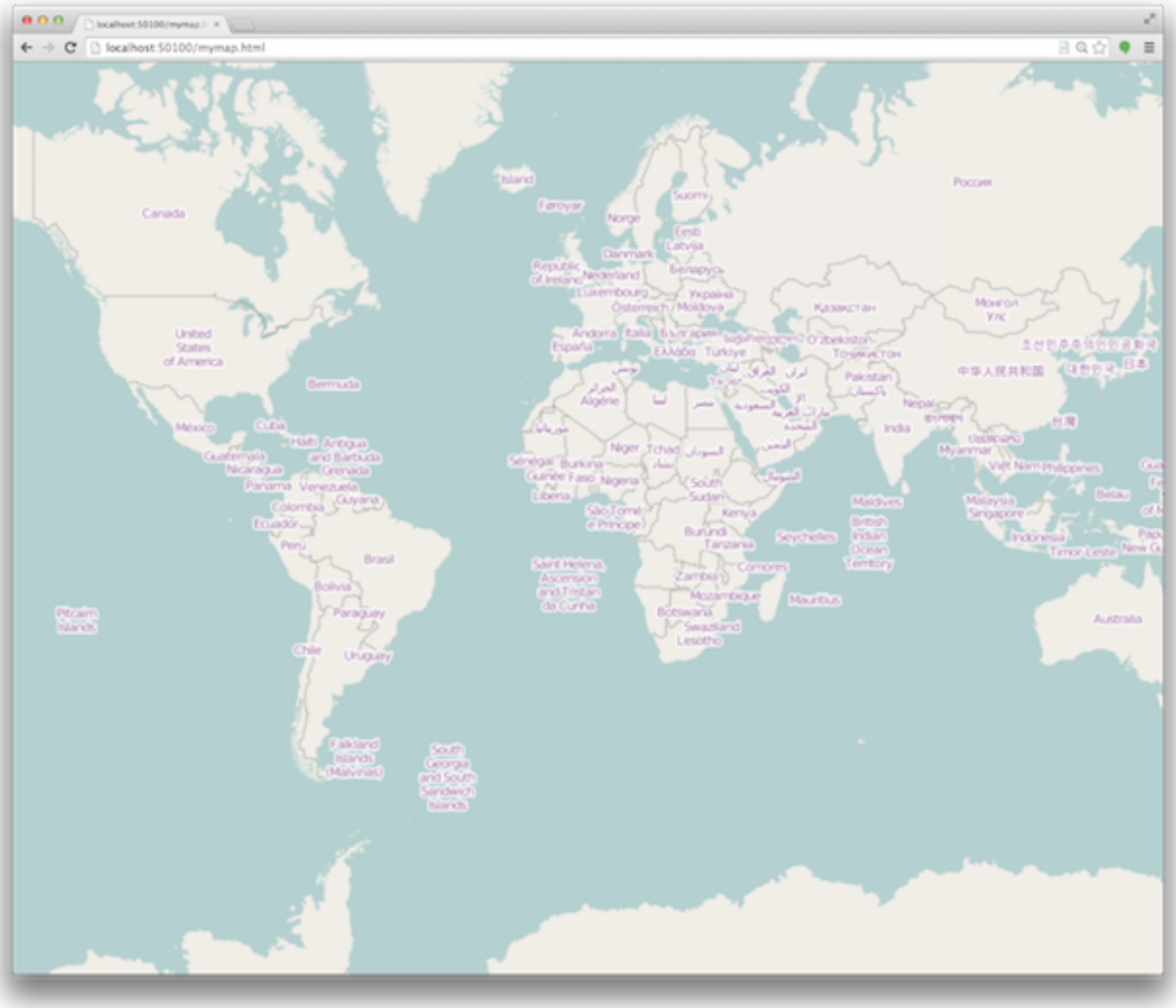
The following html gives an example of including all of the necessary files and creating a basic full map using the *osmLayer* class.

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

```
grunt serve
```

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


localhost:50100/mymap.html
Canada
United States of America
Bermuda
México
Cuba
Haiti
Antigua and Barbuda
Guatemala
Nicaragua
Grenada
Panama
Venezuela
Colombia
Guyana
Ecuador
Perú
Brasil
Bolivia
Paraguay
Chile
Uruguay
Pitcairn Islands
Falkland Islands (Malvinas)
South Georgia and South Sandwich Islands
Saint Helena, Ascension and Tristan da Cunha
Ísland
Føroyar
Norge
Suomi
Россия
Eesti
Latvija
Беларусь
Danmark
Republic of Ireland
Nederland
Luxembourg
Österreich
Україна
Moldova
Andorra
España
Italia
България
Ελλάδα
Türkiye
Қазақстан
O'zbekiston
Тоҷикистон
Монгол Улс
中华人民共和国
日本
Pakistan
India
Nepal
Myanmar
Việt Nam
Philippines
Malaysia
Singapore
Indonesia
Timor-Leste
Belau
Maldives
British Indian Ocean Territory
Seychelles
Comoros
Mauritius
Australia
Algérie
Niger
Tchad
Sénégal
Burkina Faso
Guinée
Nigeria
Liberia
South Sudan
Kenya
São Tomé e Príncipe
Burundi
Tanzania
Zambia
Mozambique
Botswana
Swaziland
Lesotho

CHAPTER 2

User's guide

2.1 Dependencies

GeoJS depends on several Javascript libraries that must be loaded prior to use as well as a few recommended libraries for optional features. As a convenience, we provide a bundle containing all required and optional dependencies in a single minified file. This bundle is built as `dist/built/geo.ext.min.js`. If you are just building a simple page out of GeoJS like in the *quick start guide*, this will probably work well; however, when using GeoJS as part of an application, you may need to customize the loading order or versions of the bundled applications. In this case, you may need to include the sources manually or bundle them yourself. The following is a list of libraries used by GeoJS.

Table 2.1: GeoJS dependencies

Library	Version	Component
jQuery	2.1	Core
proj4	2.2	Core
GL matrix	2.1	GL renderer
pnltri	2.1	GL renderer
jQuery mousewheel	3.1	Mouse interactor
d3	3.3	D3 renderer, UI widgets

Note: The versions listed are what is provided in the bundle, but other versions may work as well.

2.2 Software conventions

At it's 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. Prototypal 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) {
```

```
    // 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 overriden 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

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. Renderers can provide an extended interface so that they can be used as a *base renderer*. The base renderer provides support methods for conversion between world and screen coordinates and must respond to the map's request for navigation commands. Every map must have exactly one layer attached to a base renderer. Currently, geo.gl.vglRenderer is the only available base renderer. geo.d3.d3Renderer is also availabe for renderering features as SVG elements.

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

geo.clock The clock object is attached to the map and is resposible for maintaining a user definable concept of time. The clock can run, paused, and restarted. The clock triggers events on the map to synchronize animations.

The API documentation is in the process of being updated. You can always find the latest version at http://opengeoscience.github.io/geojs/apidocs/geo.html.

CHAPTER 3

Developer's guide

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 jshint for uniform styling and strict for common errors patterns. The style rules for geojs are located in the `.jshintrc` file in the root of the repository. These tests are preformed automatically for every file added to the build; no additional configuration is required.

3.2 Headless browser testing

Geojs uses PhantomJS for headless browser testing of core utilities. Unfortunately because PhantomJS does not support webgl at this time, it is not possible to do headless testing for any code that requires instantiating the `geo.map` class. These tests are run automatically on Travis for every pull request so they should be used whenever possible.

The headless unit tests should be placed in the `testing/test-cases/phantomjs-tests` directory. All javascript files in this directory are automatically added as test cases by CMake. They are run in PhantomJS using the Jasmine test framework. The output from Jasmine is automatically detected by the test runner, which sets it's return status to `0` if (and only if) all tests passed. You can run these tests manually in the browser by starting up a test server

```
python test/geojs_test_runner.py
```

and navigating to http://localhost:50100/test/phantomjs in your browser.

For tests that require webgl, there is a similar framework for running Jasmine unittests inside selenium. For these cases, you can add your tests inside the `testing/test-cases/jasmine-tests`. CMake will automatically pick up the scripts in the directory and generate a test case for them.

3.3 Selenium testing

Most tests for geojs require a full browser with webgl support. For these test, a framework based on Selenium is provided. This test framework is intentionally lightweight to allow for many different kinds of testing from simple Jasmine style unit tests to complicated mouse interactions with screenshot comparisons.

All selenium based tests should be placed inside subdirectories of `testing/test-cases/selenium-tests`. All subdirectories are assumed to be selenium tests by CMake and will be instrumented and run accordingly. Each subdirectory should, at a minimum, contain the following three files, which may be empty:

1. `include.css`: CSS that will be concatenated into a `style` node in the `head`.
2. `include.html`: HTML that will be concatenated into the `body`.
3. `include.js`: Javascript source that will be concatenated into a `script` node in the `head` after the inclusion of the geojs source and all dependent libraries.

Generally, developers are free to put arbitrary content into these files; however, one convention **must** be followed for the default instrumentation to work correctly. The javascript source should be wrapped in a global function called `startTest`. This function will be called automatically by the testing framework after all of the instrumentation is in place and the page is loaded. The `startTest` function will be called with function as an argument that should be called when page is ready to run the unit tests. This is provided as a convenience for the default behavior of `selenium_test.BaseTest.wait()` with no arguments. Developers can extend this behavior as necessary to provide more complicated use cases. As an example, see the `d3Animation` test case which sets a custom variable in a callback script for a test that is run asynchronously.

The compiled version of these tests are placed inside the deployment root so the users can manually see the test results. The path to each test is derived from the relative path inside `testing/test-cases/selenium-tests/`. For example, the test page in `testing/test-cases/selenium-tests/osmLayer/` is available at http://localhost:50100/test/selenium/osmLayer/ after starting the test web server.

The unit tests themselves are derived from Python's unittest module via a customized subclass `selenium_test.BaseTest`. Detailed documentation of the methods this class provides is given in the next section. Developers should feel free to extend this class with any generally useful methods as they become necessary for a wider variety test cases.

3.3.1 Example unit test

The following is a minimal example of a selenium unit test using the testing framework. More complicated examples can be found by examining the existing tests present in the source.

`hello/index.html`:

```
<div id="div-node"></div>
```

`hello/index.css`:

```
#div-node {
    text-align: center;
}
```

`hello/index.js`:

```
window.startTest = function (done) {
    $("#div-node").text("Hello, World!");
    done();
};
```

`hello/testHelloWorld.py`:

```
# Importing setupModule and tearDownModule will start up and
# shut down the web server automatically.
from selenium_test import FirefoxTest, setupModule, tearDownModule

# This test will run on firefox only.
```

```
class HelloWorld(FirefoxTest):
    testCase = ('hello', 'world')

    def test_main(self):
        # Resize the window to have consistent results.
        self.resizeWindow(640, 480)

        # Load the main html for this test directory.
        self.loadUrl('hello/index.html')

        # Wait for it to be loaded.
        self.wait()

        # Now we are ready to test the page.
        # The base class provide easy methods to test a screen shot.
        # This will take a screen shot and compare it against any
        # screenshots in the test image store at revision number 1.
        # Any failure here will raise an exception that will mark the
        # test as failed.
        self.screenshotTest('helloWorldScreenshot', revision=1)
```

3.3.2 Uploading screenshots to the image store

A script is provided in the source to help developers upload images to the data store in a way that they can be loaded automatically by the testing infrastructure. The script is built into `test/upload_test_cases.py` when selenium testing is enabled in CMake. When creating a new test (or updating a revision), the following is the recommended method for uploading test data for the example test `hello/` described above.

```
# inside the build directory
python test/upload_test_cases.py ../testing/test-cases/selenium-tests/hello
```

The script will run all the tests in this directory and prompt you if you want to upload a new image in the event that a screenshot test has failed. If you intend to start a new revision, then the revision number should be changed in the unit test source before running this script. Note: you must have write permission in the MIDAS GeoJS community before you can upload new images. Contact a community administrator for an invitation.

3.4 Code coverage

Code coverage information is accumulated automatically through custom blanketjs instrumentation when `COVERAGE_TESTS` are enabled in CMake. As long as the recommendations in this guide have been followed, all phantomjs and selenium unit tests will be instrumented for coverage reporting.

CHAPTER 4

Testing infrastructure

4.1 selenium_test

4.2 midas_handler

class midas_handler.**MidasHandler**(*MIDAS_BASE_URL='http://midas3.kitware.com/midas'*, *MIDAS_COMMUNITY='geojs'*)

Bases: object

Contains several utility function for interacting with MIDAS by wrapping api methods and caching the results.

community()

Get the id of the GeoJS community.

```
>>> midas.community()
{
    u'admingroup_id': u'121',
    u'can_join': u'1',
    u'community_id': u'40',
    u'creation': u'2014-06-02 11:38:38',
    u'description': u'',
    u'folder_id': u'11361',
    u'membergroup_id': u'123',
    u'moderatorgroup_id': u'122',
    u'name': u'GeoJS',
    u'privacy': u'0',
    u'uuid': u'538c9a7ead4a21c3b3e4e52724b3e6949487279edfad3',
    u'view': u'68'
}
```

Returns MIDAS response object.

Return type dict

getFolder(*name*, *root=None*)

Get a folder named name under root. If no root is given, use the community root.

```
>>> midas.getFolder('Testing')
u'11364'
>>> midas.getFolder('data', '11364')
u'11373'
```

Parameters

- **name** (*string*) – The folder name to find.
- **root** (*string*) – The id of the root folder.

Returns The id of the folder.

Return type string

Raises Exception If the folder is not found.

getImages(*path*, *revision*)

Download images in an item at the given path and revision.

```
>>> .getImages(('Testing', 'test', 'selenium', 'osmLayer', 'firefox', 'osmDraw.png'), 2)
[<PIL.PngImagePlugin.PngImageFile image mode=RGBA size=640x390 at 0x1019E9200>]
```

Parameters

- **path** (*tuple*) – The relative path from the community root.
- **revision** (*int*) – The item revision to download.

Returns List of Image.

Raises Exception If the path or revision is not found.

getItem(*path*, *root=None*)

Get an item at the given path. If no root is specified, use the community root.

```
>>> midas.getItem(('Testing', 'data', 'cities.csv'))
{
    u'date_creation': u'2014-06-02 15:26:12',
    ...
    u'view': u'2'
}
```

Parameters

- **path** (*tuple*) – The relative path from `root`.
- **root** (*string*) – The id of the root folder.

Returns MIDAS response object

Return type dict

Raises Exception If the item is not found.

getOrCreateItem(*path*)

Create an empty item at the given path if none exists otherwise return the item. This method will create folders as necessary while traversing the path.

Parameters **path** (*tuple*) – The relative path from the community root.

Returns MIDAS response object

Return type dict

login(*email=None*, *password=None*, *apiKey=None*)

Log into midas and return a token. If *email* or *password* are not provided, they must be entered in stdin. The token is cached internally, so the user will only be prompted once after a successful login. Alternatively, an apiKey can be provided as login credentials.

Parameters

- **email** (*string*) – The user's email address.
- **password** (*string*) – The user's password.
- **apiKey** (*string*) – The user's api key.

Return type string

Returns The login token.

`uploadFile` (*fileData*, *path*, *revision=None*)

Uploads a file to the midas server to the given path. If revision is not specified, it will create a new revision. Otherwise, append the file to the given revision number.

Parameters

- **fileData** (*string*) – The raw file contents to upload.
- **path** (*tuple*) – The relative path to the item.
- **revision** (*int*) – The revision number to append the file to.

Raises Exception If the upload fails for any reason.

Returns MIDAS response object

Return type dict

4.3 upload_test_cases

CHAPTER 5

Indices and tables

- *genindex*
- *search*

C

G

L

M

U