@maplibre/maplibre-gl-native

npm v6.0.0

node-ci

Installing

Binaries are available and downloaded during install for the following platforms:

- Operating systems:
  - Ubuntu 22.04 (amd64/arm64)
  - macOS 12 (amd64/arm64)
  - Windows (amd64)
- Node.js 16, 18, 20, 22

Run:

```
npm install @maplibre/maplibre-gl-native
```

Further platforms might work with additional libraries installed.

Testing

```
npm test
```

Rendering a map tile

The minimal example requires only the instantiation of the `mbgl.Map` object, loading a style and calling the `map.render` method:

```
var mbgl = require('@maplibre/maplibre-gl-native');
var sharp = require('sharp');

var map = new mbgl.Map();

map.load(require('./test/fixtures/style.json'));

map.render(function(err, buffer) {
    if (err) throw err;

    map.release();

    var image = sharp(buffer, {
```

```
        raw: {
            width: 512,
            height: 512,
            channels: 4
        }
    });

    // Convert raw image buffer to PNG
    image.toFile('image.png', function(err) {
        if (err) throw err;
    });
});
```

But you can customize the map providing an options object to `mbgl.Map` constructor and to `map.render` method:

```
var fs = require('fs');
var path = require('path');
var mbgl = require('@maplibre/maplibre-gl-native');
var sharp = require('sharp');

var options = {
  request: function(req, callback) {
    fs.readFile(path.join(__dirname, 'test', req.url), function(err,
data) {
      callback(err, { data: data });
    });
  },
  ratio: 1
};

var map = new mbgl.Map(options);

map.load(require('./test/fixtures/style.json'));

map.render({zoom: 0}, function(err, buffer) {
    if (err) throw err;

    map.release();

    var image = sharp(buffer, {
        raw: {
            width: 512,
            height: 512,
            channels: 4
        }
    });

    // Convert raw image buffer to PNG
    image.toFile('image.png', function(err) {
```

```
        if (err) throw err;
    });
});
```

The first argument passed to `map.render` is an options object, all keys are optional:

```
{
    zoom: {zoom}, // number, defaults to 0
    width: {width}, // number (px), defaults to 512
    height: {height}, // number (px), defaults to 512
    center: [{longitude}, {latitude}], // array of numbers
(coordinates), defaults to [0,0]
    bearing: {bearing}, // number (in degrees, counter-clockwise from
north), defaults to 0
    pitch: {pitch}, // number (in degrees, arcing towards the horizon),
defaults to 0
    classes: {classes} // array of strings
}
```

When you are finished using a map object, you can call `map.release()` to permanently dispose the internal map resources. This is not necessary, but can be helpful to optimize resource usage (memory, file sockets) on a more granular level than V8's garbage collector. Calling `map.release()` will prevent a map object from being used for any further render calls, but can be safely called as soon as the `map.render()` callback returns, as the returned pixel buffer will always be retained for the scope of the callback.

Implementing a file source

When creating a `Map`, you can optionally pass an options object (with an optional `request` method and optional `ratio` number) as the first parameter. The `request()` method handles a request for a resource. The `ratio` sets the scale at which the map will render tiles, such as `2.0` for rendering images for high pixel density displays:

```
var map = new mbgl.Map({
    request: function(req) {
        // TODO
    },
    ratio: 2.0
});
```

If you omit the `request` method, the `map` object will use the default internal request handlers, which is ok for most cases. However, if you have specific needs, you can implement your own `request` handler. When a `request` method is provided, all `map` resources will be requested by calling the `request` method with two parameters, called `req` and `callback` respectively in this example. The `req` parameter has two properties:

```
{
    "url": "http://example.com",
    "kind": 1
}
```

The `kind` is an enum and defined in `mbgl.Resource`:

```
{
    "Unknown": 0,
    "Style": 1,
    "Source": 2,
    "Tile": 3,
    "Glyphs": 4,
    "SpriteImage": 5,
    "SpriteJSON": 6
}
```

The `kind` enum has no significance for anything but serves as a hint to your implemention as to what sort of resource to expect. E.g., your implementation could choose caching strategies based on the expected file type.

The `callback` parameter is a function that must be called with two parameters: an error message (if there are no errors, then you must pass `null`), and a response object:

```
{
    data: {data}, // required, must be a byte array, usually a Buffer
object
    modified: {modified}, // Date, optional
    expires: {expires}, // Date, optional
    etag: {etag} // string, optional
}
```

If there is no data to be sent to the `callback` (empty data, or `no-content` respose), then it must be called without parameters. The `request` implementation should pass uncompressed data to `callback`. If you are downloading assets from a source that applies gzip transport encoding, the implementation must decompress the results before passing them on.

A sample implementation that reads files from disk would look like the following:

```
var map = new mbgl.Map({
    request: function(req, callback) {
        fs.readFile(path.join('base/path', req.url), function(err, data)
{
            callback(err, { data: data });
```

```
        });
    }
});
```

This is a very barebones implementation and you'll probably want a better implementation. E.g. it passes the url verbatim to the file system, but you'd want add some logic that normalizes `http` URLs. You'll notice that once your implementation has obtained the requested file, you have to deliver it to the requestee by calling `callback()`, which takes either an error object or `null` and an object with several keys:

```
{
    modified: new Date(),
    expires: new Date(),
    etag: "string",
    data: new Buffer()
}
```

A sample implementation that uses `request` to fetch data from a remote source:

```
var mbgl = require('@maplibre/maplibre-gl-native');
var request = require('request');

var map = new mbgl.Map({
    request: function(req, callback) {
        request({
            url: req.url,
            encoding: null,
            gzip: true
        }, function (err, res, body) {
            if (err) {
                callback(err);
            } else if (res.statusCode == 200) {
                var response = {};

                if (res.headers.modified) { response.modified = new
Date(res.headers.modified); }
                if (res.headers.expires) { response.expires = new
Date(res.headers.expires); }
                if (res.headers.etag) { response.etag =
res.headers.etag; }

                response.data = body;

                callback(null, response);
            } else if (res.statusCode == 204) {
                callback();
            } else {
                callback(new Error(JSON.parse(body).message));
```

```
            }
        });
    }
});
```

Stylesheets are free to use any protocols, but your implementation of `request` must support these; e.g. you could use `s3://` to indicate that files are supposed to be loaded from S3.

Listening for log events

The module imported with `require('maplibre-gl-native')` inherits from `EventEmitter`, and the `NodeLogObserver` will push log events to this. Log messages can have `class`, `severity`, `code` (HTTP status codes), and `text` parameters.

```
var mbgl = require('@maplibre/maplibre-gl-native');
mbgl.on('message', function(msg) {
    t.ok(msg, 'emits error');
    t.equal(msg.class, 'Style');
    t.equal(msg.severity, 'ERROR');
    t.ok(msg.text.match(/Failed to load/), 'error text matches');
});
```

Contributing

See DEVELOPING.md for instructions on building this module for development.