LC2 Specification

Michael Franzen, Artem Chirkin, Lukas Treyer

August 4, 2016

1 Introduction

Lightweight Urban Computation Interchange (Luci) uses GeoJSON format to represent scenario geometry. The format declares geometry information syntax, but does not declare consistent naming and geometry type mappings for Luci scenario entities, such as buildings, roads, etc. This document aims at providing guidelines on usage of Luci scenarios in Luci clients and services.

Current document may have not up-to-date action syntax specification. In order to get the most recent specification one can use "generate specification" Luci command. This document is not intended to be a replacement to that feature.

The structure of the document is organized as follows: Section 1.1 gives an information how to participate in editing of the current document. Section **??** explains the syntax of Luci actions with some commentaries. Section **??** is the main informational part of the document: it discusses the conventions used between Luci and its services.

1.1 Editing and understanding the document

The document source resides in git repository `https://bitbucket.org/treyerl/lucy.git`, the `.tex` file is `spec/lpsg/LPS-guidelines.tex`, compiled with `texlive pdflatex` tool.

The document contains a number of JSON or GeoJSON listings representing content of Luci actions. In the listings we use the following coloring scheme:

- Key names are shown in black (e.g. `action`);
- Reserved keywords, such as value types are shown in blue (e.g. `string`);
- Fixed strings constants are shown in red (e.g. `"create_scenario"`);
- Additional structural keywords are shown in grey (e.g. `OPT` means the key-value pair is optional, `XOR` before several key-value pairs means exactly one alternative).
- Comments are in purple, separated by double slash (e.g. `// comment`)

If there is a missing reserved keyword, you can add it into tex file annotation (`keywords` or `ndkeywords` lists in `lstdefinelanguage` command).

A note on jsonGeometry data type The word `geometry` in Luci specification has two different meanings. On the one hand it is a name of the key that occurs from time to time in Luci actions. On the other hand it is a name of a pre-defined data type that represents scenario geometry in JSON format. To resolve this ambiguity, in current document we use word geometry to represent the name of the key, and jsonGeometry to represent the data type. This differentiation does not introduce any changes to the existing JSON messages.

2 Luci

2.1 Structure

LC2 can be perceived as an IaaS provider, mediating between clients and *cloud* services for computational architecture analysis. Like most IaaS providers Luci's services fall into three different categories: ==storage==, ==networking== and ==computing==. We will broadly outline the main purpose of these functional units within the LC2 framework.

2.2 Storage

Luci provides functions to clients and services for storing network-wide data. However, compared to common IaaS storage systems, LC2 focusses on geospatial and geometric data. The storage is immutable, changes to the data are realized using **forward-incremental version control without synchronization** (?).

2.3 Networking

All communication in Luci is performed through a single broker, namely, the Luci server. Luci will validate the messages and, if valid, forward them to the designated service or client.

2.4 Computing

In the current state, service actions are delegated to a free computing service meeting the client-specified requirements. If there are no free computing services, the scheduler will [...]?

3 Services

3.1 Luci scenario actions

To send geometry to Luci, we wrap it in the structure that is shown in listing 1. Special type jsonGeometry wraps various types of geometry processed by Luci. It allows to enclose arbitrary number of custom-named geometries (keys KEY_NAME_N) inline (GeoJSON) or in separate files (as binary attachments).

```
{
  /* key name is an arbitrary string.
     The convention is to name it by filename of a file,
     or arbitraryname.geojson in case of GeoJSON geometry */
  KEY_NAME_1 :
    XOR { // "in-line" geometry
      format    : string // "GeoJSON", later add also TopoJSON, CurveJSON, ...
      geometry : object // GeoJSON FeatureCollection
      OPT crs   : string // name of a crs
      OPT attributeMap : { // mapping between Luci and foreign types
        LUCI_ATTRIBUTE_NAME : FOREIGN_ATTRIBUTE_NAME,
```

```
        ...
      }
    }
    XOR { // "streaminfo" - attachment description
      format      : string // shp | shx | dbf | any other file format?
      streaminfo : {
        checksum : string // MD5 sum of an attached binary data
        length : long // length of the attached binary data
        order : long // number of attachment (starts with 1)
      }
      OPT crs     : string // name of a crs
      OPT attributeMap : { // mapping between Luci and foreign types
        LUCI_ATTRIBUTE_NAME : FOREIGN_ATTRIBUTE_NAME,
        ...
      }
    },
  KEY_NAME_2 : ...,
  ...
}
```

Listing 1: structure of `jsonGeometry` data type

The type of object `geometry` is GeoJSON `FeatureCollection` – the format that is described in GeoJSON specification[1].

According to `spec/LuciSpecification.pdf`, Luci provides three operations to work with scenarios: `create_scenario`, `update_scenario`, and `get_scenario`. This document covers only GeoJSON geometry manipulation; in this format individual entities are represented as `Feature` objects in `FeatureCollection`. Each `Feature` has a property `geomID:long` that is given either by a client, or by Luci (in case if client application does not specify property `geomID`).

Creating a scenario is done via `create_scenario` action shown in listing 2. The action allows to specify a location (`projection`) and a geometry to put inside the new scenario.

```
{
  action            : "create_scenario", // constant, represents the action
  name              : string, // name of the scenario
  OPT projection    : {
    XOR crs     : string, // name of a crs
    XOR bbox    : [ [number, number]   // top-left coords [lat, long]
                  , [number, number]], // bottom-right coords [lat, long]
  },
  OPT geometry      : jsonGeometry, // wrapper around various geometry types
  OPT switchLatLon : boolean // switch lat-long to long-lat in geometry
}
```

Listing 2: JSON action structure for creating a scenario in Luci

Listing 3 shows the action to update scenario geometry. The action allows to change the name and the bounding box, as well as the geometry inside.

```
{
  action        : "update_scenario", // constant, represents the action
  ScID          : long, // ID of the scenario in Luci
  OPT name      : string, // set a new name for the scenario
  OPT bbox      : [ [number, number]   // top-left coords [lat, long]
                  , [number, number]], // bottom-right coords [lat, long]
  OPT geometry : jsonGeometry // wrapper around various geometry types
  OPT switchLatLon : boolean // switch lat-long to long-lat in geometry
}
```

Listing 3: JSON action structure for updating a scenario in Luci

[1] `http://geojson.org/geojson-spec.html`

- In order to add an entity to the scenario, one adds `Feature` into `FeatureCollection` (`geometry` object).
- In order to modify an existing entity, one must specify its property `geomID` (if given `geomID` does not exist, the entity is added to the scenario, otherwise it is edited).
- In order to delete a number of entities from the scenario, one adds an empty `Feature` into `FeatureCollection` that has a property `deleted_geomIDs:[long]` – array of `geomID` for deletion.

Listing 4 shows the action to get scenario geometry. The action allows to specify the format of the data to (Luci does transformation), and get the geometry from the scenario at given time.

```
{
  action               : "get_scenario", // constant, represents the action
  XOR scenarioname     : string // name of the scenario in Luci
  XOR ScID             : long, // ID of the scenario in Luci
  OPT format_request : string, // maybe we will change this later to "format"
  OPT crs              : string,
  OPT geomIDs          : [long], // select a subset of scenario objects
  OPT timerange        : { // time is a number - timestamp in unix format
    XOR until    : long,
    XOR from     : long,
    XOR between : [long,long],
    XOR exactly : long,
    OPT all      : boolean // include all versions (not only the last)
  }
}
```

Listing 4: JSON action structure for getting a scenario from Luci

3.2 Scenario GeoJSON geometry

Although GeoJSON specification provides all necessary geometry primitives, we need a more structured convention to define one-to-one mapping between the geometry and scenario entities. Luci does not generate an error for an input that does not follow it – the convention only describes what kind of data structures services and clients should expect.

Section **??** describes the rules assumed by Luci when communicating with all services and clients. Section **??** describes the rules assumed by most applications, but not checked in Luci. Section **??** describes the application-specific rules. Any service or client provider (Luci user) may introduce a rule that is used in their application. The providers are encouraged to add these rules into section **??**. By an agreement in our team some of the rules go up from section **??** to section **??**. In case of wide usage they might be enforced in Luci, thus moving one step up to section **??**.

3.2.1 Standardized rules

Object `geometry` in listing 1 is assumed to be of type `FeatureCollection`. Every entity is represented as a `Feature` inside that collection, and has a property `geomID:long` that is given by a client or Luci (if the client omits the property).

3.2.2 Conventional rules

Some services require 3D objects, others use only 2D footprints. To distinguish these two types of geometry, we agreed on using `Feature` property `layer`.

- Object (e.g. Building) – a 3D geometry, represented as `Feature` with `geometry` field of type `Polygon` or `MultiPolygon`. To be processed correctly by Luci services, the object requires property `layer:"buildings"`.
- Footprint – a 2D geometry, represented as `Feature` with `geometry` field of type `Polygon` or `MultiPolygon`. To be processed correctly by Luci services, the footprint requires property `layer:"footprints"`.

3.2.3 Per-application rules

Web geometry modeler The application distinguishes dynamic and static geometry: dynamic geometry can be edited, static geometry is only used for evaluation and visualization. Thus, I propose an optional property `static:boolean`. Absence of a property implies `static:false`.