Streaming CityJSON datasets

Hugo Ledoux Gina Stavropoulou Balázs Dukai

Hugo Ledoux, Gina Stavropoulou, and Balázs Dukai (2024). Streaming CityJSON datasets. Proceedings 3D GeoInfo 2024.

The code of the project is available at
https://github.com/cityjson/cjseq/
The reproducibility repository of the paper:
https://github.com/cityjson/paper_cjseq

We introduce *CityJSON Text Sequences* (CityJSONSeq in short), a format based on CityJSON and *JSON Text Sequences*. CityJSONSeq was added to the CityJSON specifications version 2.0 to allow us to stream very large 3D city models. The main idea is to decompose a CityJSON dataset into its individual city objects (each building, each tree, etc.) and create several independent JSON objects of a newly defined type: `CityJSONFeature`. We elaborate on the engineering decisions that were taken to develop CityJSONSeq, we present the open-source software we have developed to convert to and from CityJSONSeq, and we discuss different aspects of the new format, eg filesize, usability, memory footprint, etc. For several use-cases, we consider CityJSONSeq to be a better format than CityJSON because: (1) once serialised it is about 10% more compact; (2) it takes an order of magnitude less time to process; and (3) it uses significantly less memory.

1 Introduction

CityJSON is a JSON-based encoding for storing 3D city models that implements a subset of the CityGML data model version 3.0 [OGC, 2021a]. As further described in

Ledoux et al. [2019], its development started in 2017 with the aim of offering an easy-to-use and web-ready alternative to the XML-encoded CityGML files [OGC, 2023a], which in practice can be rather verbose, difficult to parse, and complex to manipulate. The first official release of CityJSON (version 1.0) was a success:

1. its JSON-based files were on average around 7 times more compact than their CityGML-XML equivalents without loss of information. See Ledoux et al. [2019] and https://www.cityjson.org/filesize/ for details, and also Praschl and Krauss [2023] for a comparison with standard computer graphics formats;
2. it was adopted as an OGC Community Standard [OGC, 2021b];
3. it was adopted by the Dutch government as a 3D standard to distribute nationwide 3D datasets;
4. several implementations and plugins have been developed, most notably FME.

However, the version 1.0 of CityJSON had one limitation: its structure for storing the coordinates of the geometries (see Figure 1) made the *streaming* of very large datasets complex, if not impossible. As further defined in Section 3, streaming refers to the possibility of downloading/transferring/processing a dataset without having to load it all in memory. Given the increasing size of datasets of 3D cities, with CityGML-XML files often exceeding 2GB, the processing of CityJSON files has become a practical challenge.

We present in this paper *CityJSON Text Sequences* (henceforth referred to as *CityJSON-Seq*), a format based on *JSON Text Sequences* [Williams, 2015] and CityJSON, and inspired by *GeoJSON Text Sequences* [Gillies, 2017]. CityJSONSeq was added to the CityJSON version 2.0 standard that was released in 2023 (and also standardised by the OGC, see OGC [2023b]). As further described in Section 4, the idea is to decompose a (often large) CityJSON file into its individual *features* (eg each `Building`, each `Bridge`, etc.) and create several JSON objects of a newly defined type `CityJSONFeature`. These objects are either serialised into a text file, or streamed from a server to client or from one application to another. This allows us to avoid a large list of vertices that need to be indexed, as it is the case with CityJSON files (see Section 2 for details).

In Section 5, we describe the open-source software we have developed to convert between CityJSON and CityJSONSeq files. We also analyse the filesizes of CityJSON and CityJSONSeq for several real-world datasets and synthetic datasets we built. It can be observed that one advantage of CityJSONSeq, besides that files containing several thousands of features can be streamed, is that it compresses further the CityJSON files by around 12%, sometimes more. We discuss in Section 5 the reasons for this interesting finding.

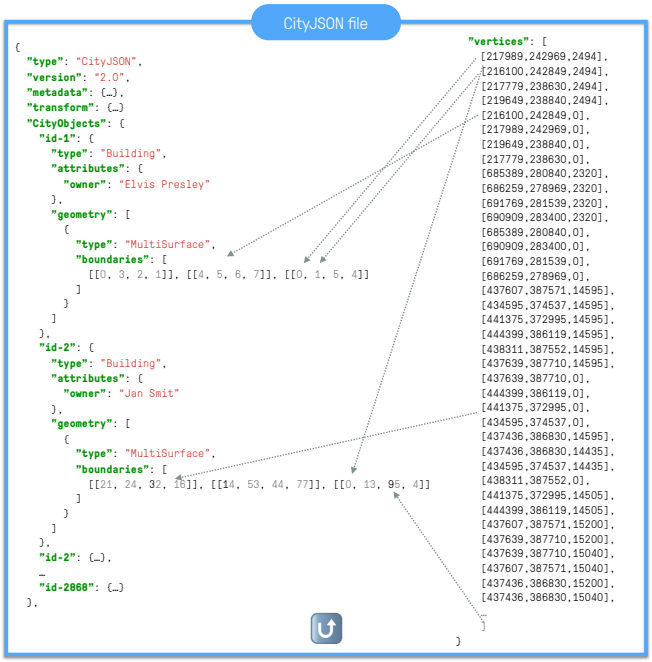


Figure 1: An example of a CityJSON file. The vertices are stored in a global list, and the position of the vertices in that list are used to represent the boundaries of the geometries (represented by the arrows, many have been left out for clarity).

2 Structure of a CityJSON file

As shown in Figure 1, a CityJSON object, which is a JSON object, represents a given geographical area, and it typically contains the following JSON properties:

1. `"type"`: it must be `"CityJSON"`;
2. `"version"`: `"2.0"` is the current version;
3. `"metadata"`: different metadata related to the dataset can be stored. The most important is the definition of the coordinate reference system (CRS).
4. `"transform"`: CityJSON `"vertices"` are compressed and stored as integers only. The parameters of this property allow us to convert from those integers back to real-world coordinates.
5. `"CityObjects"`: a dictionary where the properties are the identifiers of the city objects (*IDs*), which are any CityGML city object (for instance a `Building`, a `BuildingPart`, a `SolitaryVegetationObject`, etc.). The city objects are listed one after the other, even if some are `"children"` of others. As an example, for a `Building` containing 2 parts, the 3 objects will be represented at the same level and linked by their *IDs*, as shown in Figure 2. The schema is thus flat and all hierarchies have been removed. Each city object can have a `"parents"` and/or a `"children"` property, and this is how in the snippet the building `"id-1"` is linked to its 2 parts. The fact that a dictionary is used means that developers have direct access to the city objects through their IDs (and also in constant time if a hashmap is used to implement the dictionary while parsing the file).
6. `"vertices"`: The 3D geometric primitives in CityJSON are those of the CityGML data model, which means that multi/composite solids with several parts and/or cavities are supported. A geometric primitive does not list all the coordinates of its vertices, instead the coordinates of the vertices are stored in a separate array (the `"vertices"` property of the CityJSON object), and the geometric primitives refer to the position of a vertex in that array. This indexing mechanism has been successfully used for many years by the computer graphics community in formats as *Wavefront OBJ*[1]. There are several advantages to this approach. First, the files can be compressed: 3D vertices are often shared by several surfaces, and repeating them can be costly, especially if they are very precise (sub-millimetre precision is often used). Second, this approach increases the topological relationships that are explicitly stored in the file, and several operations (eg determining building adjacency) can be sped up and made more robust. Third, it is very easy to convert all coordinates to a representation listing; the inverse is not true. However, this list of vertices is the reason why the streaming of geometries is problematic, since in practice it can contain several millions vertices. To be able to reconstruct

[1] https://en.wikipedia.org/wiki/Wavefront_.obj_file

```
"CityObjects": {
  "id-1": {
    "type": "Building",
    "attributes": {...},
    "children": ["id-2", "id-3"],
    "geometry": [{...}]
  },
  "id-2": {
    "type": "BuildingPart",
    "parents": ["id-1"],
    "geometry": [{...}]
    ...
  },
  "id-3": {
    "type": "BuildingPart",
    "parents": ["id-1"],
    "geometry": [{...}]
    ...
  }
  ...
  "id-77": {}
}
```

Figure 2: CityJSON mechanism to flatten out the schema: the city objects are stored in a flat list, and they are linked together with the properties `"parents"` and `"children"`.

a single Building, all the `"vertices"` need to loaded in memory, which can mean waiting for millions of unused vertices to be deserialised.

7. `"appearances"`: Both textures and materials for surfaces are supported. The material of a surface is represented with the X3D specifications[2], and for the textures the COLLADA specifications[3] are reused.
8. `"geometry-templates"`: Geometry templates are geometries defined once and reused by applying a translation, a rotation, and/or a scaling. They are mostly used for city objects like trees, bus stops, and lamp posts.

3 Streaming (3D) datasets

In the context of geo-information, a *stream* is a sequence of data that is available over a period of time, and "can be thought of as items on a conveyor belt being processed one at a time rather than in large batches"[4].

For GML-based formats (which are feature-centric), modifying a file for the purpose of streaming is usually a simple task that involves sending the features in the dataset

[2]https://en.wikipedia.org/wiki/X3D
[3]https://www.khronos.org/collada/
[4]From https://en.wikipedia.org/wiki/Stream_(computing)

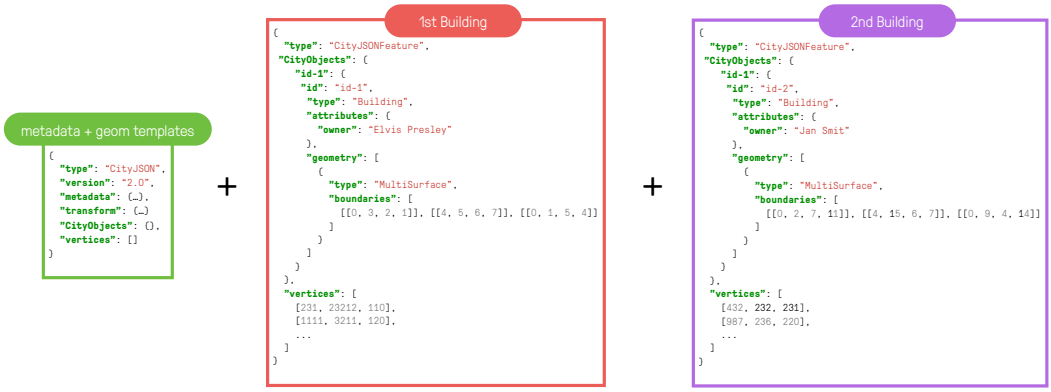


Figure 3: The CityJSONSeq of a CityJSON dataset with two buildings contains three JSON objects: one for the metadata, plus one for each building.

one-by-one. However, notice that this is only true for GML files that follow the Simple Feature paradigm [OGC, 2006]. For more complex data models like CityGML, where geometry templates and *XLinks* are used, streaming often requires a large amount of pre-processing. *XLinks* are links between elements in a file (similar to *pointers* in a computer program), a concrete example in a CityGML file is a cube that lists the geometries of 5 of its surfaces, but its 6th surface is simply a link to another surface somewhere else in the file (which belongs to another building for instance). If the linked surface has not be seen in the stream yet, the receiver cannot process the cube and needs to wait for that specific 6th surface to appear (eg to calculate its volume). It is not always possible to (re-)order a CityGML file so that all the references can be resolved without having to store extra information until it appears in the stream. However, it is always possible to resolve the *XLinks* before streaming a dataset (that is, in our example, copy the geometry of the 6th surface to the cube), but this means that the filesize will increase, and that converting back to the original file will not be possible.

For the *GeoJSON* format [Butler et al., 2016], which follows the Simple Feature paradigm, creating a stream is trivial since each of the features in the dataset becomes one JSON object serialised to one line [Gillies, 2017]. There are no links possible between features, each JSON object is independent.

For formats that use a global indexing of vertices, such as CityJSON and most formats used for storing meshes in computer graphics (eg OBJ and STL), the reorganisation of the elements in a file is more complex but nonetheless possible. Isenburg et al. [2003] describe algorithms and tools that interleave the vertices and faces in a file (instead of having one large list of vertices at the end) and add simple tags to inform that specific vertices are not used anymore in the stream (and thus can be freed from memory). This allows us, in theory, to process/edit/manipulate infinitely large meshes, since they never have to be completely loaded in memory. The idea is exemplified by the construction of gridded terrains that are gigabytes in size [Isenburg et al., 2006]. However,

```
{
  "type": "CityJSONFeature",
  "id": "id-1",
  "CityObjects": {
    "id-1": {
      "type": "Building",
      "attributes": {
        "roofType": "gabled roof"
      },
      "children": ["mybalcony"],
      "geometry": [...]
    },
    "mybalcony": {
      "type": "BuildingInstallation",
      "parents": ["id-1"],
      "geometry": [...]
    }
  },
  "appearance": {...}
  "vertices": [...]
}
```

Figure 4: An example of a `CityJSONFeature` for a Building with a balcony referenced in its `"children"` property.

this cannot be implemented in CityJSON directly because all the vertices need to be listed in the JSON property `"vertices"`.

4 CityJSON Text Sequences

As shown in Figure 3, a CityJSONSeq decomposes a CityJSON object into its features to create a sequence of several JSON objects. Those JSON objects are of type `CityJSONFeature`, which allows the storage of a single feature, for instance a `Building`, together with its "children" objects (eg a `BuildingPart` and/or a `BuildingInstallation`). Each feature is independent, it has its own list of vertices (which is thus *local* to the JSON object, and is usually rather small, see next section for details) and its own textures and materials (if any). The allowed properties are shown in Figure 4; notice that the `"id"` property is used to clearly identity the "parent" of the feature, in case there are children.

CityJSONSeq follows the specifications of `ndjson` (newline delimited JSON)[5] and two constraints are added for handling CityJSON:

1. each JSON Object must conform to the *JSON Data Interchange Format specifications* [Bray, 2017] and be written as a UTF-8 string;
2. each JSON Object must be followed by a new-line (LF: `"\n"`) character, and it may be preceded by a carriage-return (CR: `"\r"`);

[5] https://github.com/ndjson/ndjson-spec/

```
{"type":"CityJSON","version":"2.0","transform": {"scale":[1.0,1.0,1.0],"
  translate":[0.0, 0.0, 0.0]},"metadata":{"referenceSystem":"https://www.opengis
  .net/def/crs/EPSG/0/7415"},"CityObjects":{},"vertices":[]}
{"type":"CityJSONFeature","id":"id-1","CityObjects":{...},"vertices":[...]}
{"type":"CityJSONFeature","id":"id-2","CityObjects":{...},"vertices":[...]}

```

Figure 5: An example of a CityJSONSeq stream containing 3 features.

3. a JSON Object must not contain the new-line or carriage-return characters;
4. the first JSON Object must be of type `CityJSON`;
5. the following JSON Objects must be of type `CityJSONFeature`.

Note that a `CityJSONFeature` object does not contain all the information that is required for reconstructing the feature. Most commonly, the `"transform"` property, the CRS, and the `"geometry-templates"` must be known in order to correctly reconstruct and georeference the city objects. The rule #4 ensures that those are available. The CityJSON object must contain a `"transform"` property and eventually the other properties if needed; `"CityObjects"` and `"vertices"` must be present but they must be empty (to ensure that the JSON object is valid).

The CityJSONSeq for Figure 3 is shown in Figure 5.

5 Experiments with real-world datasets

To convert between CityJSON and CityJSONSeq files (and vice-versa), we have developed the open-source software *cjseq*, which is available at https://github.com/cityjson/cjseq/ under a permissive open-source license. The command-line program handles the conversion not only of the geometries, but also of the materials, the textures, and the geometry templates that the dataset could contain. It includes three sub-commands:

1. `cat`: CityJSON $\rightarrow$ CityJSONSeq;
2. `collect`: CityJSONSeq $\rightarrow$ CityJSON;
3. `filter`: to filter city objects in a CityJSONSeq, randomly or based on a bounding box.

It should be observed that the conversion is an efficient process: the rather large dataset *Helskinki* from Table 1, which contains more than 77 000 buildings and whose CityJSON file is 572 MB, takes only 4.7 sec to be converted to a CityJSONSeq file, and the reverse operation takes 5.7 sec (on a standard laptop).

Table 1: The datasets used for the benchmark.

	dataset		size of file			vertices		
	CityObjects	app.[a]	CityJSON	CityJSONSeq	compr.[b]	total	largest[c]	shared[d]
3DBAG	1110 bldgs		6.7 MB	5.9 MB	12%	82 509	4112	0.1%
3DBV	71 634 misc		378 MB	317 MB	16%	4 110 319	116 670	21.0%
Helsinki	77 231 bldgs		572 MB	412 MB	28%	3 038 576	2202	0.0%
Helsinki_tex	77 231 bldgs	tex	713 MB	644 MB	10%	3 038 576	2202	0.0%
Ingolstadt	55 bldgs		4.8 MB	3.8 MB	25%	87 972	12 800	0.0%
Montréal	294 bldgs	tex	5.4 MB	4.6 MB	15%	31 585	3393	2.0%
NYC	23 777 bldgs		105 MB	95 MB	10%	1 035 804	2608	0.8%
Railway	50 misc	tex+mat	4.3 MB	4.0 MB	8%	73 554	14 966	0.4%
Rotterdam	853 bldgs	tex	2.6 MB	2.7 MB	-4%	22 246	631	20.0%
Vienna	307 bldgs		5.4 MB	4.8 MB	11%	47 220	2025	0.0%
Zürich	52 834 bldgs		279 MB	247 MB	11%	3 472 989	4069	2.6%

[a] appearance: 'tex' is textures stored; 'mat' is material stored

[b] compression factor is $\frac{size(CityJSON)-size(CityJSONSeq)}{size(CityJSON)}$

[c] number of vertices in the largest feature of the stream

[d] percentage of vertices that are used to represent different city objects

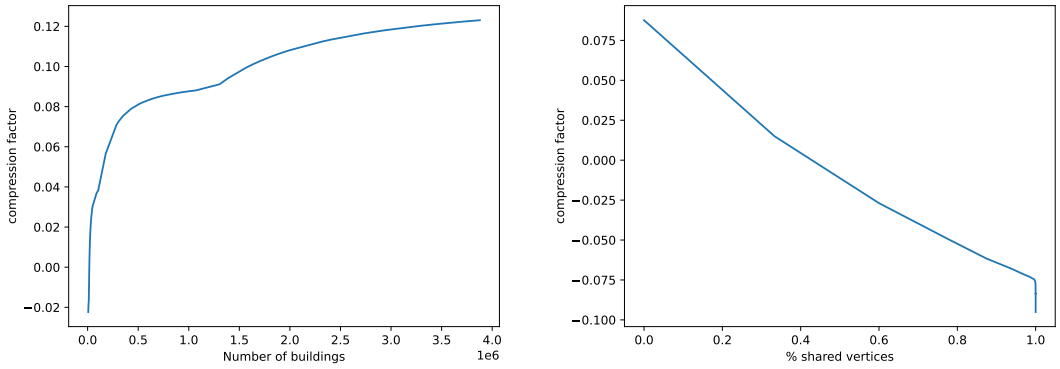
5.1 Filesize comparison

We have converted with *cjseq* several publicly available files, and Table 1 shows an overview of the files stored both in CityJSON and CityJSONSeq. The files are available in the reproducibility repository of the paper[6].

First observe that—contrary to intuition—the filesize of a dataset serialised as a CityJSONSeq file is around 12% compacter than serialised as a CityJSON file, and in the case of *Helsinki* it is 28%. An even larger compression factor is noted in most datasets whose texture, materials, semantics and attributes have been removed. The main reason for this is that the indices of the vertices are low integers for each feature (because the lowest index in each feature is always "0" and is incremented by 1 until the total number of vertices), and they do not increase to very large integers in contrast to the vertices in CityJSON. For instance, the dataset *Helsinki* contains a total of more than 3 millions vertices, but its largest feature contains only but 2202 vertices. The fact that many indices are used for representing the geometries (and the textures) means that if several large numbers are used then the filesize will grow; if the maximum vertex index is around 2000 for each feature then the filesize will be reduced.

Only one dataset sees its filesize slightly increase, by 4%, when serialised to a CityJSONSeq file: *Rotterdam*. The reasons for the increase (or decrease) are many, and we discuss in the following the 3 most relevant: (1) the total number of vertices; (2) the number of shared vertices; (3) the presence of textures.

[6] https://github.com/cityjson/paper_cjseq



(a) (b)

Figure 6: Compression factor of CityJSONSeq files for different synthetic datasets. **(a)** Based on the number of buildings; the buildings are stored as simple cuboids that are randomly generated. **(b)** Based on the percentage of shared vertices; the same cuboid buildings are used, but we position them adjacent to each others to create shared vertices.

Number of vertices. If a dataset has few vertices, as it is the case with *Rotterdam*, then the indices will not be large integers and this might not be favourable for the compression. As an experiment, we have created around 100 synthetic CityJSON datasets containing buildings, and each building is represented as a simple cube, which is randomly generated. There are no attributes, no semantics, and no textures/materials. Figure 6a shows that, as the CityJSON filesize increases, the compression factor increases. The smallest file contains only 526 buildings and its compression factor is -2% (thus CityJSONSeq has a larger filesize than that of CityJSON), while the largest file has 3 960 105 buildings, and a compression factor of more than 12%.

Shared vertices. The number of shared vertices between different city objects also influences the compression factor. Shared vertices are those used to represent walls incident to two adjacent buildings. In CityJSON they are conceptually the same vertices and each of the surfaces refer to them, but in CityJSONSeq they have to be listed separately in each of the buildings. It should be said that most of the datasets have very few vertices that are shared (most have less than 2%, except 2 datasets have around 20%, *Rotterdam* being one of them). To understand the correlation between the compression factor and the percentage of shared vertices in a datasets, we have modified the script to generate random cuboid buildings: the distribution of the buildings is not random, we have enforced that several buildings are adjacent to others (so that they share vertices with other buildings). The relationship between the compression and the percentage of shared vertices can be seen in Figure 6b for around 100 datasets containing exactly 1 000 000 buildings. If the number of shared vertices is 0% this means that we have

Table 2: Comparison of the processing time and maximum RAM usage for processing CityJSON and CityJSONSeq files. The resident set size (RSS) is used, which is the portion of main memory occupied by the Python script.

	RAM used (MB)		**time (s)**		
	CityJSON	CityJSONSeq	CityJSON	CityJSONSeq	diff
3DBAG	76.9	16.1	0.10	0.07	1.4X
3DBV	4101.8	123.8	10.95	3.59	3.1X
Helsinki	3743.1	15.0	13.39	2.74	4.9X
Helsinki_tex	5004.8	19.1	29.60	4.72	6.3X
Ingolstadt	65.5	21.3	0.08	0.06	1.3X
Montréal	79.3	20.8	0.11	0.07	1.6X
NYC	949.5	16.0	1.78	0.70	2.5X
Railway	69.6	29.6	0.09	0.07	1.3X
Rotterdam	42.4	14.6	0.04	0.04	1.0X
Vienna	60.1	15.7	0.06	0.05	1.2X
Zurich	2793.1	16.3	6.05	2.00	3.0X

1 000 000 buildings that are disconnected; in this case we obtain a compression factor of around 8% (as was the case in Figure 6a). If all the buildings are adjacent to another one (thus nearly 100% of the vertices are shared), then we can see that the compression factor is about -10% (which means that the size of the CityJSONSeq file is larger than that of the CityJSON file).

Textures. It should also be noticed that the attributes attached to city objects, as well as the semantics attached to surfaces, have no influence on the compression factor since they are local to each city object. However, we can state that textures have an influence on the compression factor. See for instance the dataset *Helsinki* and its counterpart *Helsinki_tex* (which is the same the same geometries and attributes, only the textures were removed). The dataset with textures has a compression of 10% while the one without 28%. This is explained by the fact that the `"textures"` property must be used for each feature, while in a CityJSON object they are all stored at only one location. Since textures can be used by several features (all the bricks of a building could use the same one), this means that often the same properties for textures are copied to several features.

5.2 Processing speed comparison

We compare the speed and memory footprint of accessing each city object in a dataset. This operation is common in applications that manipulate 3D city models.

When a city model is stored in its entirety in one CityJSON object, we need to deserialise the whole CityJSON object into memory in order to access the `"transform"` and `"vertices"` properties for instance.

With a CityJSONSeq file, we can read the file line by line, processing and discarding the city objects one by one (and thus never have in memory more than the city object itself and the first JSON object in the stream). As shown in the experiments below, this allows for very efficient operations in terms of both CPU and memory usage.

We have processed all the datasets from Table 1 with two simple Python scripts that iterate through the city objects and their geometries, and increment a global counter for the geometry type (`Solid` or `MultiSurface`) and report it at the end. This is just an example of a simple local operation, any other operation such as calculating the area of the façades or counting the number of windows could have been performed. The results for both the maximum memory footprint and the time used are shown in Table 2. The scripts are available in the reproducibility repository of the paper[7].

Notice that the dataset *3DBV* contains not only buildings but also the terrain, and a few large areas are stored as a triangulation containing a very large amount of vertices; this is the reason why the maximum memory use is larger than for other datasets.

The results in Table 2 indicate that there is a significant benefit to using CityJSONSeq over CityJSON, at least for operations that do not require analysing or processing city objects that are close to each other. Operations like calculating volumes, merging and subsetting files, finding city objects with specific attributes, etc. will all use significantly less memory and will be significantly faster. Operations like modifying the CRS or updating the metadata would not even require to loop through the features, just to alter the first object in the file. Operations like calculating the surface of shared walls [Agugiaro et al., 2022] are however not suitable for streams.

6 Discussion and future work

While CityJSONSeq was developed mostly for streaming large 3D city datasets, the fact that it has a much lower memory footprint and that it takes an order of magnitude less time to process (for some local operations) makes it an attractive alternative to CityJSON for several use-cases.

It should be noticed that the CityJSON specification does not prescribe the storage of CityJSONSeq, only the structure of a CityJSONSeq stream. In practice, CityJSONSeq can be stored in a variety of ways, for instance in a single file, each feature in a separate file, in a database, etc. The optimal storage solution depends on the implementing application. As a concrete example, CityJSONSeq is used by *cjdb* [Powałka et al., 2024], an importer/exporter tool that stores one feature per row in PostgresSQL. Additionally,

[7] https://github.com/cityjson/paper_cjseq

in order to facilitate pagination, CityJSONSeq is the return format of the 3DBAG API[8], which contains all 10 million buildings in the Netherlands with detailed roofs [Peters et al., 2022].

From the point-of-view of practitioners, we should stress that CityJSONSeq can easily be processed with *Unix pipes* (also called *pipelines*). Pipelines allow us to chain several processes together, the output of a process becomes the input of the next one, and so on. Given 2 processes, the 2nd one can usually start before the 1st one has finished processing all the data. This means that, in practice, the processing of a dataset can be performed by writing several programs that perform a few small tasks; this makes the development and maintenance of code simpler. Moreover, it allows us to write the code in different languages. The pipelines used for preparing the datasets used in this paper were actually a mix of Python (the program *cjio* and *cjdb*) and Rust (*cjseq* and *cjval*), and another program written in C++ could easy be added.

Finally, because the structure of CityJSONSeq is nearly the same as that of CityJSON, in practice adding support for CityJSONSeq requires a minimum amount of effort since most of the code to parse and/or generate CityJSON objects can be reused. We have already added support for CityJSONSeq in a few of the open-source tools[9] and we will continue in the future.

References

G. Agugiaro, A. Zwamborn, C. Tigchelaar, E. Matthijssen, C. León-Sánchez, F. van der Molen, and J. Stoter. On the influence of party walls for urban energy modelling. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVIII-4/W5-2022:9–16, 2022.

T. Bray. The javascript object notation (JSON) data interchange format. Technical Report RFC 8259, Standard from Internet Engineering Task Force (IETF), 2017.

H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and S. Hagen. The GeoJSON Format. Technical Report RFC 7946, Standard from Internet Engineering Task Force (IETF), 2016.

S. Gillies. GeoJSON Text Sequences. Technical Report RFC 8142, Standard from Internet Engineering Task Force (IETF), 2017.

M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. In *IEEE Visualization, 2003. VIS 2003.*, pages 465–472, 2003. doi: 10.1109/VISUAL.2003.1250408.

[8] https://api.3dbag.nl/
[9] https://www.cityjson.org/software/

M. Isenburg, Y. Liu, J. R. Shewchuk, J. Snoeyink, and T. Thirion. Generating raster DEM from mass points via TIN streaming. In *Geographic Information Science—GIScience 2006*, volume 4197 of *Lecture Notes in Computer Science*, pages 186–198, Münster, Germany, 2006.

H. Ledoux, K. A. Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis. CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards*, 4(4), 2019. doi: 10.1186/s40965-019-0064-0.

OGC. OpenGIS implementation specification for geographic information—simple feature access. Open Geospatial Consortium inc., 2006. Document 06-103r3.

OGC. OGC City Geography Markup Language (CityGML) Part 1: Conceptual Model Standard. Open Geospatial Consortium inc., 2021a. Document 20-010, version 3.0.0, available at https://docs.ogc.org/is/20-010/20-010.html.

OGC. CityJSON Community Standard 1.0. Open Geospatial Consortium inc., 2021b. Document 20-072r2, version 1.0.

OGC. OGC City Geography Markup Language (CityGML) Part 2: GML Encoding Standard. Open Geospatial Consortium inc., 2023a. Document 21-006r2, version 3.0.

OGC. CityJSON Community Standard 2.0. Open Geospatial Consortium inc., 2023b. Document 20-072r5, version 2.0.

R. Peters, B. Dukai, S. Vitalis, J. van Liempt, and J. Stoter. Automated 3D reconstruction of LoD2 and LoD1 models for all 10 million buildings of the Netherlands. *Photogrammetric Engineering and Remote Sensing*, 88(3), 2022.

L. Powałka, C. Poon, Y. Xia, S. Meines, L. Yan, Y. Cai, G. Stavropoulou, B. Dukai, and H. Ledoux. In T. Kolbe, A. Donaubauer, and C. Beil, editors, *Recent Advances in 3D Geoinformation Science (Proceedings of the 18th 3D GeoInfo Conference)*, pages 781–796. Springer, 2024. doi: https://doi.org/10.1007/978-3-031-43699-4_47.

C. Praschl and O. Krauss. Extending 3D geometric file formats for geospatial applications. *Applied Geomatics*, 16(1):161–180, 2023.

N. Williams. JavaScript Object Notation (JSON) Text Sequences. Technical Report RFC 7464, Standard from Internet Engineering Task Force (IETF), 2015.