CITYJSON + WEB = NINJA

S. Vitalis[1]*, A. Labetski[2]*, F. Boersma, F. Dahle, X. Li, K. Arroyo Ohori, H. Ledoux, J. Stoter

3D Geoinformation Group, Delft University of Technology, the Netherlands -
[1] s.vitalis@tudelft.nl, [2] a.labetski@tudelft.nl

KEY WORDS: Visualisation, 3D City Modelling, CityJSON, Web application, Versioning

ABSTRACT:

As web applications become more popular, 3D city models would greatly benefit from a proper web-based solution to visualise and manage them. CityJSON was introduced as a JSON encoding of the CityGML data model and promises, among several benefits, the ability to be integrated with modern web technologies. In order to provide an implementation of a web application for CityJSON data, that can be used as a reference for other applications, we developed `ninja`. It is a web application that allows the user to easily load and investigate a CityJSON model through a web browser. In addition, it offers support for a complex feature of CityJSON: the experimental versioning mechanism. In this paper, we describe the motivation, requirements, technical aspects and achieved functionality of `ninja`. We believe that such a web application can facilitate the adoption of 3D city models by more practitioners and decision makers.

1. INTRODUCTION

As web applications become increasingly popular, 3D city models' usage would greatly benefit from the existence of such applications with support for 3D city model files. That is because the ability to view, investigate, and edit 3D city models through a web browser, without the need to first convert them to other formats or to use specialised tools, could greatly simplify their usage for domain experts and decision makers.

CityGML is a commonly studied data model to represent 3D city models (Open Geospatial Consortium, 2012), and it is also a GML encoding for the storage and exchange of such data. While the CityGML data model has been extensively used in academia (Giovanella et al., 2019; Park et al., 2019; Braun et al., 2018), there is limited software support for the data format (Noardo et al., 2019). One of the main reasons for this is GML's complexity and verbosity which makes implementation challenging for software developers. More specifically, for web applications, CityGML files must be first loaded into a database and converted to other intermediate formats (e.g. 3D Tiles) in order to make their web dissemination possible (Yao et al., 2018), which results in a complicated and time-consuming process.

In order to solve this issue, CityJSON[1] was introduced as a JSON encoding for the CityGML data model (Ledoux et al., 2019). It focuses on maintaining the majority of features of CityGML through a simpler file structure that can be easier mapped to modern programming languages' data structures. One of the main reasons for implementing a JSON encoding was the fact that it is considered today the most used information exchange mechanism for web applications[2].

CityJSON's web-friendly encoding promises to be a solid foundation for implementing such web applications. We developed `ninja`[3] as a reference implementation of a web application that can handle CityJSON files. We set the following requirements regarding the application's functionality: 1) a 3D geometric viewer with support for the main geometric types (from ISO 19107:2003: Geographic information—Spatial schema (2003)); 2) a clean way to investigate the model's semantics (e.g. city objects' hierarchy and attributes); 3) an easy way to show and edit information for specific city objects; and 4) a straightforward viewer for versioning information stored in a dataset, based on the data structure proposed by Vitalis et al. (2019).

*Corresponding author

2. BACKGROUND

2.1 CityGML

CityGML is the "Open" Geospatial Consortium's standard for the representation, storage, and exchange of 3D city models (Open Geospatial Consortium, 2012). It is the name of both the data model and the XML encoding, which is an important distinction to emphasise as it pertains to CityJSON. CityGML consists of 13 modules which include `Buildings`, `Bridges`, `Transportation`, `Relief`, etc. CityGML also uses the concept of Level of Detail (LoD) which indicates the level of abstraction that a model has in comparison to its real-world counterpart (Biljecki et al., 2016).

In the realm of 3D model visualisation there exist several frameworks to support the web distribution of common graphics formats such as COLLADA, OBJ or X3D; but direct support for CityGML is limited (Blut et al., 2019). There are several commercial software packages for CityGML visualisation, including ArcGIS (ESRI), Bentley Map (Bentley Systems) and the CityEditor (3DIS GmbH) and freeware/open source software such as FZKViewer (KIT Karlsruhe) (Blut et al., 2019). Nevertheless, there is no direct support for CityGML visualisation and dissemination through the web, without the use of additional tools (such as 3DcityDB) and intermediate formats (such as GLTF) (Ledoux et al., 2019; Noardo et al., 2019).

2.2 CityJSON

CityJSON is the JSON encoding for the majority and most-used features of the CityGML data model (Ledoux et al., 2019). It

is easier to read and renders the files more compact (on average six times more compact than their CityGML equivalents). All of the CityGML modules are mapped to CityJSON objects, but for the sake of simplicity and efficiency, some modules and features were omitted and/or simplified[4]. Furthermore, several features that are absent in CityGML are included in CityJSON. These include: a refined concept for the LoDs (Biljecki et al., 2016) and structured support for metadata, which is lacking in CityGML. Table 1 summarises the extent of software support for visualising, editing, generating, etc. CityJSON files, including the focus of this paper, `ninja`.

2.3 Versioning for 3D City Models

Versioning is utilised for the management of changes in information, and is well established in the realm of computer programming where it can be used to track changes in source code (Spinellis, 2005). In the realm of 3D city models, new versions are regularly created due to the fact that cities themselves are constantly changing, the modelling aspect of a project may change, or certain maintenance processes may cause changes to a dataset (Vitalis et al., 2019).

There is no mechanism to manage different city model versions in the current version of CityGML (2.0), but there exists a proposal for CityGML 3.0 as developed by Chaturvedi et al. (2017). The proposal has versions as aggregations of timestamped features, which also store the reason for the transition as well as the type. At the same time, the proposed CityGML versioning module conflates versioning with lifecycle modelling, which adds complexity to the solution by addressing two different issues simultaneously. More specifically, this is due to the fact that the module implies that the order of real-life operations matters, although that is outside the context of versioning. In addition, there is currently no software implementation for the versioning mechanism of CityGML files.

Vitalis et al. (2019) proposed and implemented a framework that focuses specifically on solving the problem of storing the versions of multiple city objects in a semantic-agnostic way. In their methodology, all versions of all objects are stored in a single file where a CityJSON file acts as a repository, referred to as "versioned CityJSON" (vCityJSON). A vCityJSON file has a structure that is similar to a regular CityJSON file, simply with the addition of a "versioning" property which also contains version metadata. There is also an open versioning prototype[19] CLI that supports many common versioning functions including drawing a log graph of commits, finding the difference between commits, creating and merging branches, etc.

3. REQUIREMENTS

We have developed `ninja` with the intention to provide a reference implementation for a web application using the CityJSON file format. From a technical standpoint, our purpose was to evaluate the suitability of CityJSON for modern web technologies, such as JavaScript front-end frameworks. To make the application relevant for practitioners, we focused on certain points of usability.

First, the application must be able to show all common geometric types found in real-world datasets. Those are, basically, all surfacic and volumetric types (i.e. `MultiSurface`, `Solid` and their composite counter-parts). That functionality includes the ability to display inner boundaries (i.e. holes for surfaces and inner-shells for solids).

Second, we wanted `ninja` to provide a comprehensive and clean way for the user to easily perceive the semantic aspects of the city model. This means that the user must be able to understand the structure of the model based on the hierarchical relationship between the city objects (parent and child). In addition, we want the application to present basic information for every city object (i.e. its object type and geometric types) in a concise way.

Third, there must be a way to view as much detailed information as possible for an individual city object, if prompted by the user. All basic properties of a city object (e.g. the id, type, and parent/children objects) and its attributes must be accessible to the user upon request. This also includes the ability for the user to access and edit the raw JSON information of the city object.

Finally, the application must have some basic support for versioning. When a file contains information about multiple versions, as described in Section 2.3, then it must be possible to see the history of the city model and select a specific version to investigate.

4. IMPLEMENTATION

`ninja` is written in `JavaScript` using the `Vue`[20] framework. We chose `Vue` for two reasons: first, it allows for the encapsulation of logic in reusable components, which reduces code redundancy and enables flexibility; and second, it promotes the use of data binding, which significantly improves the quality of code reducing the amount of code required for the manipulation of data in a web application.

In order to make `ninja` responsive to variable screen sizes we built its user interface through `Bootstrap`[21]. For 3D visualisation purposes, we chose to use `threejs`[22] due to its simplicity and comparably solid performance when used in web applications.

4.1 Architecture

The application is composed of a main `Vue` component, which acts as an orchestrator for multiple individual components (such as those described in Section 4.2). The main component stores the CityJSON data and holds the information of the application's state in memory. It passes the information to the individual components, which offer respective interactions to the user.

4.2 CityJSON Vue components

`ninja` is composed of `Vue` components that offer individual functionality with CityJSON data. We have isolated those components in a library, so that they can be used to develop other web applications. This library is distributed through the `npm` registry[23] so that other developers can easily access it.

4.2.1 ThreeJsViewer is a 3D viewer implemented in `threejs`. The component takes as input a complete city model (in CityJSON) and is responsible for rendering its geometries in 3D. For this, a triangulation of the prescribed boundary surfaces of every geometry is computed, which is done using the `earcut`[24] library. The component can track a user's selection of

		Software	Viewer	Generator	Editor	Transformer	Parser	Validator	Versioning*
○	▷	3dfier[5]		♦					
●	▷	azul[6]	♦						
●	▷	Blender[7] plugin[8]	♦			♦			
○	⋈	citygml-tools[9]				♦			
○	◁	citygml4j[10]					♦		
○	⋈	cjio[11]				♦	♦	♦	
●	▷	FME[12] reader/writer[13]	♦	♦					
●	▷	`ninja`	♦		♦				♦
●	▷	QGIS[14] plugin[15]	♦						
◐	⋈	val3dity[16]						♦	
○	⋈	Versioning Prototype[17]							♦
●	▷	Web-viewer[18]	♦						

*Experimental

○ - Command Line Interface (CLI) ; ● - Graphical User Interface (GUI) ; ◐ - Both CLI & GUI
▷ - Application ; ◁ - Library ; ⋈ - Both Application and Library

Table 1. Software Support for CityJSON.

a geometry, which can be invoked by a parent component in order to handle the selection for its purposes. For instance, `ninja` uses this event so that when it is triggered, the respective object's information is then shown on the screen (see Section 5.3).

4.2.2 CityObjectsTree is a tree view component which shows the hierarchical structure of the city objects in a city model. The main input of the component is an array of city objects. The component builds the tree structure, where first-level objects are the items of the array provided and their children are shown as leaves. Every object is shown with the symbology, as described in Section 5.2.

4.2.3 CityObjectCard is a card component which shows details of a city object and allows the editing of its JSON data. The component takes as input a city object and is responsible for rendering its details, as described in Section 5.3. It also offers an option to allow editing for the specific object, which the parent component can enable through a respective property. When changes to JSON are saved, an event is triggered that can be used by the parent component to act accordingly (e.g. to refresh the information of other components).

5. FUNCTIONALITY

In this section we iterate through the four main aspects of functionality that we focused on in the development of `ninja`.

5.1 Visualisation

Visualisation in `ninja` is provided through the respective 3D viewer, as can be seen in Figure 1. The viewer shows all geometries of every object and supports all ISO 19107 surfacic and volumetric geometric types (`MultiSurface`, `CompositeSurface`, `Solid`, `MultiSolid`, `CompositeSolid`). The inner boundaries of surfaces and solids are fully supported. The geometries are coloured according to the respective city object's type (e.g. `Building` or `Terrain`). The mapping between city object types and colours can be altered by the user through the settings of the application. The user can, also, navigate through the model using typical functions (pan, zoom, and rotate) and can double click to select an object for further information (see Section 5.3).

5.2 Model semantics investigation

The left sidebar in `ninja` provides a clean way to interpret the semantic aspect of the city model (see Figure 1). Mainly, this is done through the tree-view, which visualises the hierarchical relationships between the city objects. The first-level city objects are listed as root nodes and their children city objects can be collapsed/expanded accordingly.

For each city object, there is a symbology that is associated with its type (e.g. `Building`, `Terrain`, `Transportation`). A symbol also appears to indicate the LoD of the geometries of an object or whether it utilises a geometric template. The tree-view can be filtered using the respective search bar, where users can search for objects based on their id, type, or attributes. Finally, the user can select a city object by double-clicking on it to show further information (see Section 5.3).

5.3 Displaying and editing object information

Upon selecting a city object through the 3D viewer or the tree-view, `ninja` displays the details of the selected city object. This is done through an info box which displays all semantic information of the object. This includes: the city object's type and its respective symbology; the id of the selected object; and the list of attributes or geometries of the object.

Through the respective button, the user can switch to editing the object. This is done by allowing the user to alter the city object's raw JSON data (see Figure 2). When the data is saved, the new data is stored in memory in the browser, while the tree-view and 3D viewer are refreshed automatically to reflect the changes. The edited version can then be easily downloaded and saved for future use.

5.4 Versioning

`ninja` supports parsing versioning information for files created through the versioning prototype, as described in Section 2.3. When a CityJSON file that contains versions is loaded in `ninja`, a second sidebar option is enabled for interacting with the different versions.

The versioning sidebar lists the branches in the file through a drop-down list, and the list of versions that compose the selected branches history, as can be seen in Figure 3. For each

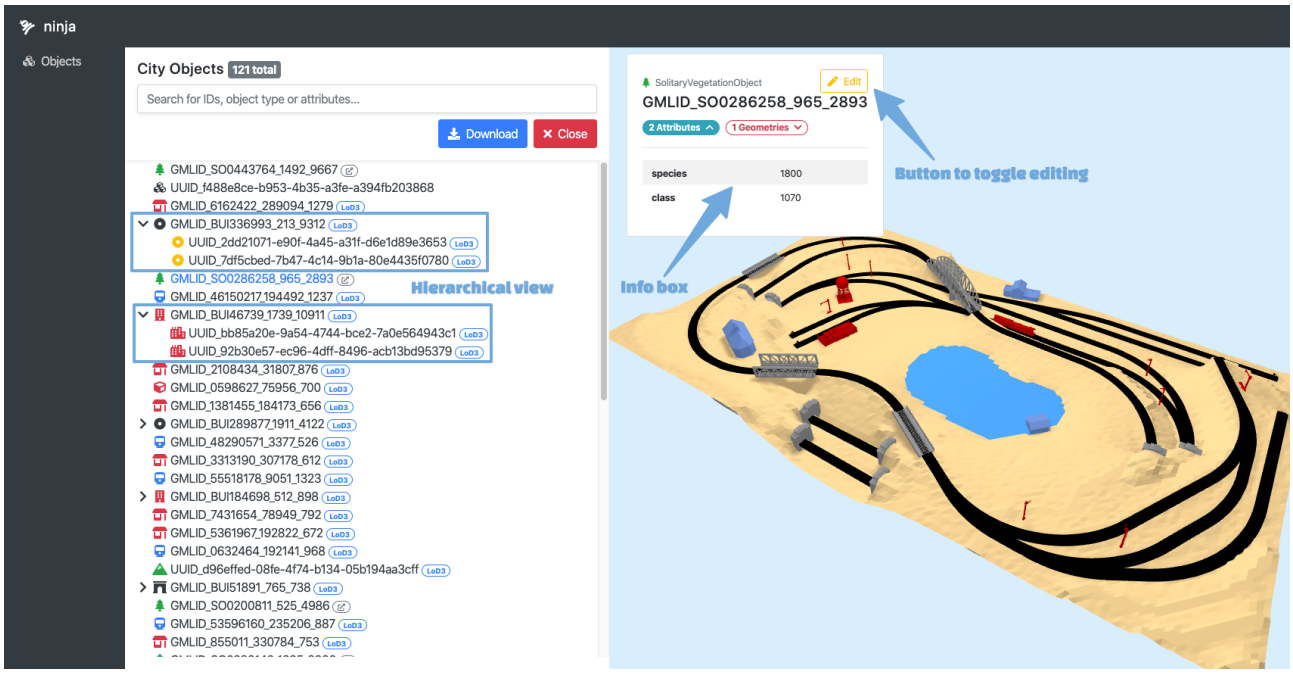

Figure 1. Once a CityJSON file is loaded, the user will see the viewer on the right and the tree-view on the left.

version, the main metadata is shown: the author, the time of submission, and the version's commit message. The list can be used to select or download an individual version. When a version is selected, the tree-view and 3D view is updated to show only the version-specific objects.

6. DISCUSSION

We have developed `ninja` to provide a reference implementation for web software using CityJSON. This process has allowed us to further evaluate the suitability of CityJSON for web development, which was our initial goal. We can confirm that manipulating a JSON-encoded 3D city model was a simple process in JavaScript, and that building a relatively complex application to manipulate 3D city models was possible. Unlike with an XML-based encoding, no additional libraries were required to manipulate and edit the model. We would like to point out that `ninja` is by itself an achievement for 3D city modelling, as there is no JavaScript library to parse/edit/manipulate XML-encoded CityGML files.

Nevertheless, certain details of the CityJSON specifications could be further refined to make the implementation simpler. From our implementation, we have realised that the use of a global list of vertices is not necessarily ideal for applications where geometry and semantics should be bound together. This is because a global vertex list can only benefit the rendering process when all geometries are handled as one big mesh. This is not efficient, though, when city objects need to exist as individual meshes in order to preserve their semantics.

Regarding the use of a browser as the underlying platform of a complete 3D city model tool, we have identified both advantages and disadvantages. From a user's perspective, having to use a simple browser in order to navigate through a model is a significant benefit. Nevertheless, using a browser to process big datasets can be problematic, given that JavaScript and other related web technologies are bound to the performance limitations that their nature implies (i.e. high-level programming languages, garbage collection, etc.). As expected, when bigger or more complex data models needed to be handled by `ninja`, the responsiveness of the interface and the performance of the 3D viewer would deteriorate. That is further underpinned by the specific implementation of the browser's JavaScript engine: from our tests, V8 (used by Google Chrome) had a noticeably better performance than SpiderMonkey (used by Mozilla Firefox), for example.

From a technical perspective, we found certain aspects of web development easier, in company with CityJSON, than they would have been with native applications. Especially through the use of assisting frameworks, such as `Vue` and `Bootstrap`, building a complete and feature-rich user interface was a relatively easy task. However, certain elements of data manipulation would require more intricate development techniques in order to ensure that there are no performance trade-offs. For example, two-way data binding had to be explicitly avoided, in certain cases, to obviate the problem of reloading the whole city model when small changes were made.

While designing the user interface, we concluded that there are certain limitations to the amount of information we could present without causing confusion to the user. While showing more information in a structured and well-formatted way was not a technical issue, users sometimes prefer to have less information about the model displayed.

Incorporating the prototype versioning mechanism in `ninja` was done deliberately in order to assess the potential use of such a concept. We can conclude that certain aspects of versioning manipulation can be easily implemented with web technologies, such as parsing the versioning metadata, showing the list of versions, and extracting a version as CityJSON through JavaScript. Nevertheless, implementing more complex functions, such as comparing individual objects and showing differences between versions is more difficult than in programming languages with more powerful mechanisms for data processing (e.g. Python).

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-VI-4-W1-2020-167-2020 | © Authors 2020. CC BY 4.0 License.

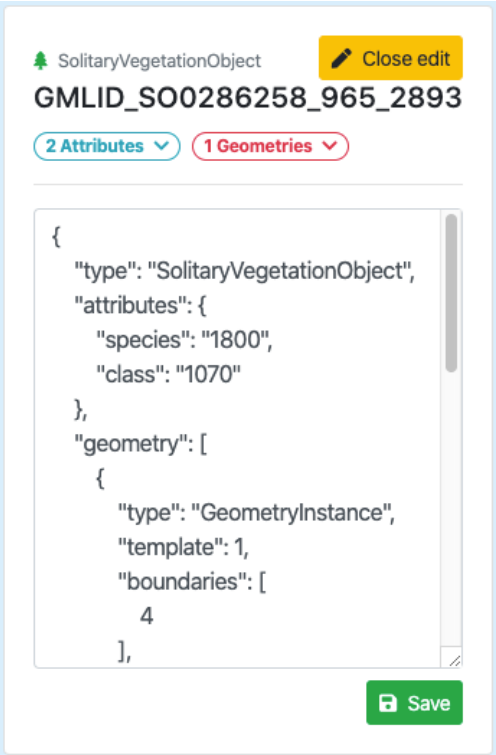

Figure 2. With `ninja`, it is possible to edit the raw JSON and save it via the info box on an object-by-object basis.

Due to the architecture of the application, it is possible to easily swap components in the future. For instance, currently the viewer is based on `threejs`, but it could be easily possible to have another implementation for other 3D libraries (e.g. Cesium).

We believe `ninja` can be used extensively by researchers, practitioners, and decision makers as a tool to easily access and investigate a 3D city model. That is due to its balance between simplicity and the amount of information displayed to the user. Furthermore, we believe that showing versioning information through a web user interface can further assist in convincing the community about the usefulness and added value of such functionality.

7. FUTURE WORK

Work in `ninja` is ongoing and in an open development. We host everything on a GitHub repository[25] and we encourage users to report back to us on how we can continue to make `ninja` better. We discuss below some ideas that are either already in development or at least on our horizon.

We are actively working on adding support for geometry instances/templates, points, and lines, as well as the ability to visualise textures (Figure 4). For files that contain multi-LoD representations of the same objects, a way to handle these cases will need to be explored. Users will also be able to have even more control over visualisation by extending the 3D view colouring schemes to include categories beyond the object type, e.g. by semantic surface instead.

We also plan to allow `ninja` to load 3D city models not only by uploading a file, but by accessing an *OGC API—Features* compliant web service (Open Geospatial Consortium, 2019). This is the newer and completely revamped standard formally known as a WFS, which is a RESTful service. We are currently working on a best practices document for CityJSON to be served as a WFS (this requires a few modifications given that in CityJSON some properties like "vertices" and "textures" are global), and we will test it first with `ninja`. One advantage that we see from this is the capability to *stream* large datasets.

In a similar fashion, to address scalability, we are planning to add support for loading multiple CityJSON files in ninja. More importantly, we are interested in investigating the use of metadata as the main definition file to define a tileset of CityJSON files that could be loaded in parallel. The main use case we would like to tackle, in this case, is visualisation and management of tiled datasets.

In the future the tree-view will exist independently from the viewer. It will have further tools that will assist in understanding the semantic data better. The ability to group objects by module will be one such tool. Filtering the tree-view based on module, attribute, or relationship (parent, child) will be a further tool. This means that users who wish to only interact with the tree-view are also not forced to load the viewer.

Given the importance of metadata for understanding information about the entire model, there will be support for viewing the metadata of a model and inserting/editing it. This will also be organised as a tree-view. Further metadata can also be added to the info boxes of individual features with information about the presence or absence of textures and/or materials.

It is also important to emphasize that developing `ninja` will always be about finding an equilibrium between effectiveness, usability, and simplicity. We will continue to focus specifically on balancing the editing capabilities versus the versioning capabilities.

ACKNOWLEDGEMENTS

We would like to thank Balázs Dukai and Ravi Peters for their insightful remarks after extensively testing `ninja`.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 677312 UMnD).

A. SOFTWARE VERSION

At the time of writing this paper this is the status of the versions:

`ninja` - 0.3

`cityjson-vue-components` - 0.3.2

References

Biljecki, F., Ledoux, H., Stoter, J., 2016. An improved LOD specification for 3D building models. *Computers, Environment and Urban Systems*, 59, 25–37.

Blut, C., Blut, T., Blankenbach, J., 2019. CityGML goes mobile: application of large 3D CityGML models on smartphones. *International journal of digital earth*, 12(1), 25–42.

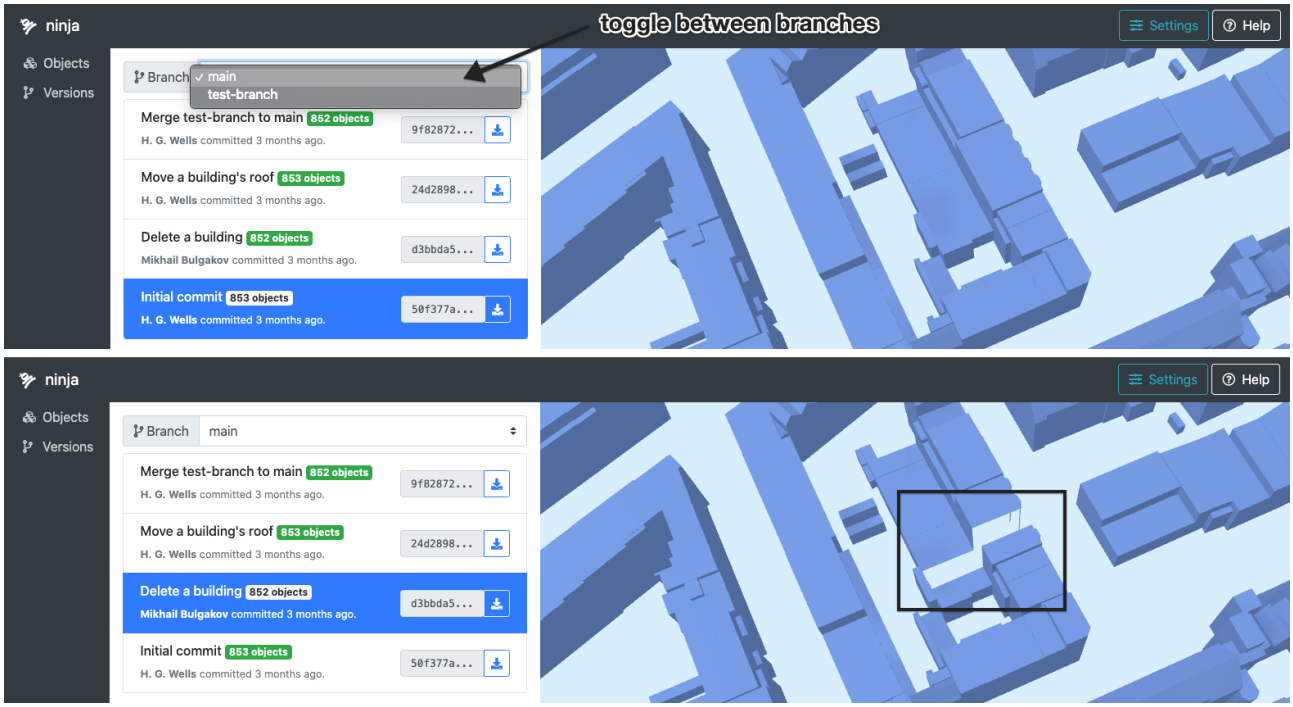

Figure 3. An example of versioning in `ninja`. The image at the top demonstrates the ability to toggle between different branches. It also has the the model at the state in which it was first versioned. The image below shows a different version of the model, in this case a building was deleted.

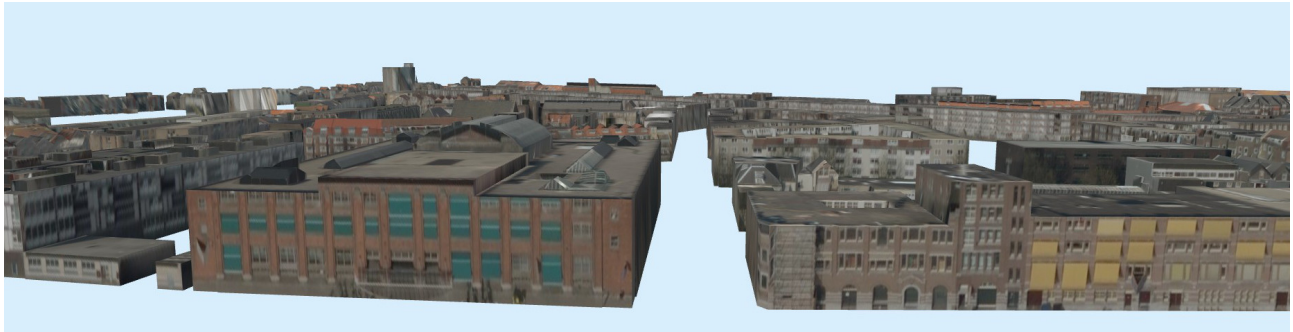

Figure 4. Demo of our work-in-progress support for textures in `ninja`'s 3D viewer.

Braun, R., Weiler, V., Zirak, M., Dobisch, L., Coors, V., Eicker, U., 2018. Using 3D CityGML models for building simulation applications at district level. *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, IEEE, 1–8.

Chaturvedi, K., Smyth, C. S., Gesquière, G., Kutzner, T., Kolbe, T. H., 2017. Managing versions and history within semantic 3D city models for the next generation of CityGML. *Advances in 3D Geoinformation*, Springer, 191–206.

Giovanella, A., Bradley, P. E., Wursthorn, S., 2019. Evaluation of topological consistency in CityGML. *ISPRS International Journal of Geo-Information*, 8(6), 278.

ISO 19107:2003: Geographic information—Spatial schema, 2003. International Organization for Standardization.

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

Noardo, F., Arroyo Ohori, K., Biljecki, F., Krijnen, T., Ellul, C., Harrie, L., Stoter, J., 2019. GeoBIM benchmark 2019: Design and initial results. G. Vosselman, S. J. Oude Elberink, , M. Y. Yang (eds), *ISPRS Geospatial Week 2019*, International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XLII-2number W13, ISPRS, 1339–1346. ISSN: 2194–9034 (Internet and USB), 1682–1750 (Print), 1682–1777 (CD-ROM).

Open Geospatial Consortium, 2012. OGC City Geography Markup Language (CityGML) Encoding Standard 2.0.0.

Open Geospatial Consortium, 2019. OGC API—Features—Part 1: Core. Document 17-069r3, version 1.0.

Park, S. H., Jang, Y.-H., Geem, Z. W., Lee, S.-H., 2019. CityGML-Based Road Information Model for Route Optimization of Snow-Removal Vehicle. *ISPRS International Journal of Geo-Information*, 8(12), 588.

Spinellis, D., 2005. Version control systems. *IEEE Software*, 22(5), 108–109.

Vitalis, S., Labetski, A., Arroyo Ohori, K., Ledoux, H., Stoter, J., 2019. A Data Structure to Incorporate Versioning in 3D City Models. *14th 3D GeoInfo Conference 2019*, ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, IV-4number W8, ISPRS, 123–130.

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., Kolbe, T. H., 2018. 3DCityDB — a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, 3(1).

Notes

1. https://www.cityjson.org
2. https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html
3. https://ninja.cityjson.org
4. https://www.cityjson.org/citygml-compatibility
5. https://github.com/tudelft3d/3dfier
6. https://github.com/tudelft3d/azul
7. https://www.blender.org
8. https://github.com/cityjson/Blender-CityJSON-Plugin
9. https://github.com/citygml4j/citygml-tools
10. https://github.com/citygml4j
11. https://github.com/cityjson/cjio
12. https://www.safe.com
13. https://github.com/safesoftware/fme-CityJSON
14. https://qgis.org
15. https://github.com/cityjson/cityjson-qgis-plugin
16. https://github.com/tudelft3d/val3dity
17. https://github.com/tudelft3d/cityjson-versioning-prototype
18. https://viewer.cityjson.org
19. https://github.com/tudelft3d/cityjson-versioning-prototype
20. https://vuejs.org
21. https://getbootstrap.com
22. https://threejs.org
23. https://www.npmjs.com/package/cityjson-vue-components
24. https://github.com/mapbox/earcut
25. https://github.com/cityjson/ninja

Revised July 2020