Delft University of Technology

azul: A fast and efficient 3D city model viewer for macOS

Arroyo Ohori, G.A.K.

DOI
10.1111/tgis.12673

Publication date
2020

Document Version
Final published version

Published in
Transactions in GIS

Citation (APA)
Arroyo Ohori, G. A. K. (2020). azul: A fast and efficient 3D city model viewer for macOS. *Transactions in GIS*, *24*(5), 1165-1184. https://doi.org/10.1111/tgis.12673

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

DOI: 10.1111/tgis.12673

RESEARCH ARTICLE

Transactions in GIS WILEY

azul: A fast and efficient 3D city model viewer for macOS

Ken Arroyo Ohori

3D geoinformation, Delft University of Technology, Delft, The Netherlands

Correspondence
Ken Arroyo Ohori, 3D geoinformation, Delft University of Technology, Delft, The Netherlands.
Email: k.ohori@tudelft.nl

Abstract

3D city models are an important research topic within geographic information, but there is still a lack of good tools to work with them in practice. In an attempt to alleviate this problem and to help with our own research, we have developed azul, a free and open-source macOS 3D viewer that was especially engineered for the visualization of 3D city models. The aim of this article is, first of all, to describe the inner workings of azul as a complete methodology to efficiently visualize 3D city models, and which can be also applied to other hierarchically structured data models, which are becoming more prevalent in geographic information. In addition, the article has three ancillary goals: (a) to present other general-purpose methods that are useful to efficiently process 3D city models (e.g., robust error-correcting parsers and data models that can be used with multiple formats); (b) to describe technical issues and problematic aspects related to current 3D city model formats that are known by developers but are not properly documented in the scientific literature; and (c) to foster an open discussion about the best data structures and algorithms to process 3D city models in practice.

1 | INTRODUCTION

More open 3D models of cities are becoming available,[1] and these are being used in an increasing number of applications (Biljecki, Stoter, Ledoux, Zlatanova, & Çöltekin, 2015). Some examples that use 3D city models on

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.
© 2020 The Authors. *Transactions in GIS* published by John Wiley & Sons Ltd

their own include: analyzing a building's potential for the installation of solar panels (Jakubiec & Reinhart, 2013; Liang, Gong, Li, & Ibrahim, 2014), the areas viewable from its windows (Bartie, Reitsma, Kingham, & Mills, 2010; Peters, Ledoux, & Biljecki, 2015) and the shadows cast by it (Nguyen & Pearce, 2012). In addition, other kinds of data can be combined with 3D city models to create new applications, such as building information models for building permit issuing (Noardo, Ellul, et al., 2019) and geologic data for infrastructure planning (Diakité & Stoter, 2017).

Many software tools are used to create such 3D models and to implement the applications built on them, but among these tools, having a good viewer is paramount. When working with a 3D city model, many common tasks are often best done in an interactive visual environment, including getting an understanding of the distribution, attributes and geometric complexity of the objects in the model; finding the elements that objects are composed of; and comprehending the various geometric errors that are present in the model (cf. Ledoux, 2018). In addition, having a good open-source viewer is crucial for research related to 3D city models, as it allows anyone to modify the code to see the new features that are created when devising and testing new 3D city model standards, data sets and methodologies.

Unfortunately, there is a lack of good free and open-source viewers for 3D city models on all operating systems, and particularly so on macOS. Generic 3D model viewers and 3D modeling software work well only for 3D city models in standard 3D modeling formats, such as OBJ, PLY or OFF, which lack the meaningful semantics that are available in 3D city model formats. Moreover, since these tools do not support 3D city model formats directly, when this support is added externally (e.g., through plug-ins), they still lack ways to view and analyze the hierarchies and rich semantics that are present in these formats. Commercial GIS software, such as FME, provides better support for semantic 3D city models, but their price limits who can afford to use them and their closed nature limits how they can be used. For example, it is not possible to test variations of a new format for research in a novel application. Recently, support for 3D city model formats has been added to (open-source) QGIS through plug-ins as well, but its relational data model and associated viewer are not an ideal fit for the hierarchical nature of 3D city model formats. Finally, other 3D city model viewers are unfortunately Windows-only, so they can only be used in other operating systems through virtual machines and compatibility layers, which comes with an increased use of resources, reduced performance and awkward workflows that clash with those in the remainder of the system.

Solving the lack of a good 3D city model viewer for Mac with a tool that could help with our research at the 3D geoinformation group was the main driver behind the development of *azul*, a free and open-source macOS 3D viewer that was especially engineered for the visualization of 3D city models. In addition to good Mac support, we had a few other important software design goals based on the main problems we have found when working with 3D city models: (a) support for all the 3D city model formats that we regularly use in our research; (b) fast loading and display of models; (c) sufficiently low memory use to smoothly display large areas at once; (d) robust methods that gracefully handle the typical errors present in most models; and (e) intuitive interaction that takes into account the particularities of 3D city models (e.g., a large horizontal and small vertical extent) and those of standard Mac hardware (e.g., multi-touch trackpads rather than mice with multiple buttons and scroll-wheels). These goals should also be built on top of a mostly C++ base for future cross-platform compatibility. Moreover, there was a strong motivation to show that open-source academic software can implement some of the more ambitious implementation techniques that are often present in commercial software but usually absent in more research-oriented software, such as very fast low-level parsers, low-overhead graphics application programming interfaces (APIs), and single instruction, multiple data (SIMD) instructions.

The remainder of this paper is organized as follows. Section 2 presents background information about 3D city model formats. Section 3 describes other available 3D city model viewers. Section 4 presents the methodology used by azul to parse, load and visualize models. Section 5 covers implementation details. Section 6 contains results and an evaluation of the performance of azul. Section 7 presents a discussion and conclusions.

2 | BACKGROUND

2.1 | Semantic 3D city models

3D city models are a representation of the environment in and around a city using three-dimensional geometries. They have an emphasis on buildings, but they also model other kinds of urban features, such as roads, railways, water bodies, vegetation, terrain and city furniture (e.g., traffic lights and benches). For each of these classes, specific semantic information is defined (e.g., addresses for buildings), which are then attached to relevant surfaces. Note that this is somewhat different from the volumetric representations used in other models (e.g., building information models).

Internally, 3D city models follow a very hierarchical representation (Figure 1). For instance, a model might have a building, which is divided into building parts, where a part has a specific exterior, which has a wall, which has a window, which has a specific geometry, and so on. This fact makes them different from many other types of geographic information that follow a flatter relational approach, such as a data set where every element has the same tuple of attributes.

The main formats that are used to represent city models are: CityGML in its default GML encoding; CityJSON; and simpler formats originating from 3D modeling computer graphics. These are described in the following subsections.

2.2 | CityGML

CityGML (OGC,) is a both a data model and a specific encoding of the data model based on GML 3.1.1 (OGC, 2004). The data model explains the proper way to recursively decompose a geographic region into semantically meaningful types, while the encoding defines the specific hierarchy of XML tags, elements and attributes that should be used when the model is stored in a `.gml` or `.xml` file.

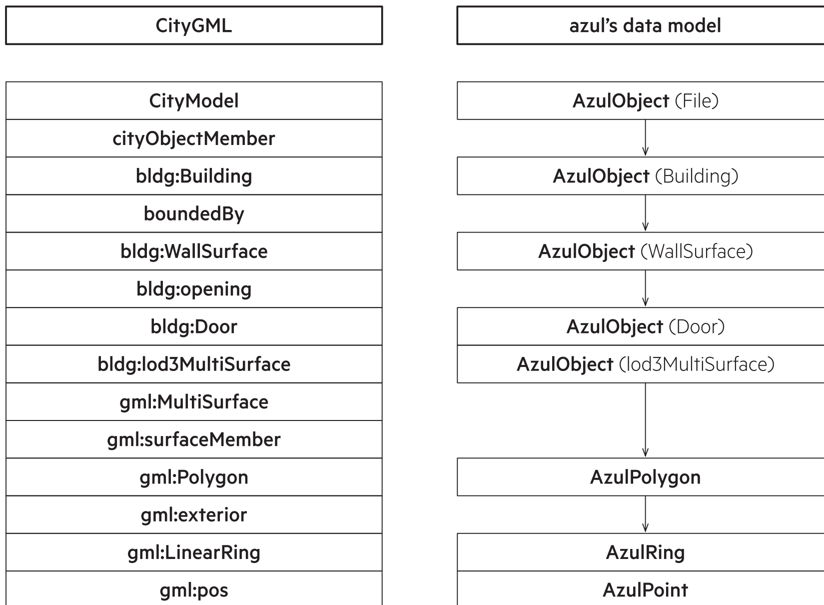


FIGURE 1 Obtaining the coordinates of an example point in a 3D city model stored in CityGML requires traversing a complex hierarchy of classes (left). The hierarchy is somewhat simplified in azul's internal data model (right). Note that in a relational model, every level in the hierarchy might involve a separately modeled table

The CityGML data model focusses primarily on modeling buildings, which have rich semantics, with transportation-related features as a second priority (roads, railways, tunnels, bridges and other transportation areas). Many of these can be decomposed into surfaces with specific semantics (e.g., walls, windows, doors and roofs for buildings). The standard also defines other kinds of features (water bodies, vegetation, land use and terrain) in a more haphazard manner, which are usually used to fill in the gaps between the built features. In practice, this usually means water bodies for everything from large lakes to small ditches, vegetation for both large forested areas and small road medians, land use for most other remaining areas when there is some sort of meaningful classification available, and terrain for everything else.

Other relevant features of CityGML include: support for up to five levels of detail (LoDs) representing different geometries for each object, reuse of geometries using a custom transformation for each (known as *implicit geometries*), textures and materials, a series of code lists for each feature type, as well as an extension mechanism known as application domain extensions (ADEs).

In addition to the data model, the CityGML standards also specify an XML-based encoding, where the geometries are encoded using a subset of GML 3.1.1 using only linear geometries (e.g., `gml::Point`, `gml::Surface` and `gml::Solid`), as well as their aggregates (i.e., `Multi`…) and composites (i.e., `Composite`…).

From an implementation perspective, CityGML in its default GML encoding can be very challenging. Many of its difficult features are inherited from XML, including its complexity and large size, non-relational structure that is hard to normalize, difficulty of parsing and difficulty of refactoring. Other difficult features come from GML, such as the possibility to encode the same geometry in an extremely high number of alternative ways (http://erouault.blogspot.com/2014/04/gml-madness.html), and the use of potentially many coordinate and reference systems (CRS) using very complex arbitrary URIs. To these, CityGML itself adds a verbose and very deeply nested structure (extending even into object attributes) and the widespread use of XLinks (https://www.w3.org/TR/2005/NOTE-xlink10-ext-20050127/) (e.g., to link semantic objects to their geometries).

2.3 | CityJSON

CityJSON (https://www.cityjson.org) (Ledoux et al., 2019) is another 3D city model format that is based on the CityGML 2.0 data model. Broadly, it is thus able to store the same city models as CityGML's GML encoding, but it is a more user- and developer-friendly JSON-based encoding that does away with most of the problematic features of CityGML's GML encoding.

Most of the characteristics of CityGML are also present in CityJSON: the same feature types with a focus on buildings and transportation, multiple levels of detail, materials and textures, and the same geometry types originating from GML. However, it is significantly easier from an implementation perspective because it has a much more compact and flatter encoding, it has simpler CRS handling, it has no XLinks to resolve, geometries are defined much more consistently, and its extension mechanism (CityJSON Extensions) is built in a way that does not require special support for each ADE.

There are a few noteworthy features that differ from CityGML: (a) a separate list of vertices that is used throughout (as in several computer graphics formats); (b) a transform object that defines a transformation that is applied to all vertices; (c) more complex metadata handling; and (d) a slightly different method to compute geometry templates (equivalent to implicit geometries in CityGML).

2.4 | 3D modeling formats

For research related to 3D city models, the formats that support 3D city model semantics are most relevant. However, simpler formats originating from computer graphics and 3D modeling are easier to work with and have

wider software support, and they are thus often used during their creation and processing. The (Wavefront) OBJ, POLY and OFF formats are some examples, all of which consist of a vertex list with coordinates, and a list of faces that refer to these vertices by their indices in the list. OBJ and OFF are used in 3D modeling software. POLY is the format used by Triangle (https://www.cs.cmu.edu/quake/triangle.html) (Shewchuk, 1996) and TetGen (https://www.wias-berlin.de/software/index.jsp?id=TetGen&lang=1) (Si, 2015), which is used with triangulated and tetrahedralized models.

3 | RELATED WORK

There are different kinds of software that can be used to view 3D city models. When the models are in standard 3D modeling formats, a very large number of generic 3D model viewers can be used. Among these, MeshLab (http://www.meshlab.net) (Cignoni et al., 2008) is likely the most widely used option. It is open source, cross-platform, loads almost all models, exhibits good performance and has a number of editing and mesh processing features that are useful for editing 3D city models.

A related option is to use 3D modeling software such as Blender (https://www.blender.org), which is also open source, cross-platform, can be used to generate high-quality rendered images and has a CityJSON plug-in under development (https://github.com/cityjson/Blender-CityJSON-Plugin). Other free and open-source options include FreeCAD (https://www.freecadweb.org) and K-3D (http://www.k-3d.org). All the main commercial 3D modeling software also supports these formats, including 3ds Max and Maya.

On macOS, there is also some basic native support for formats like OBJ in the operating system, which is usable in QuickLook or in applications such as Preview and Xcode. Similarly, there is support for OBJ from Windows 10 in included applications such as the 3D Builder and the Mixed Reality Viewer. However, it is important to note that both 3D viewers and 3D modeling software generally do not support 3D city model formats, and even when this support is added through plug-ins, they lack ways to view and analyze the hierarchies and rich semantics that are present in these models.

In addition, there are a number of web viewers for CityJSON, including the unnamed viewer on the CityJSON website (https://viewer.cityjson.org) and ninja (https://ninja.cityjson.org), the latter of which was built for experiments on versioning using CityJSON. Unfortunately, there seems to be no similar web viewer for CityGML, and the complexity of dealing with the complex encoding in JavaScript makes it unlikely to exist in the near future. In fact, the CityGML website has been broken since at least 2018, so efforts to create a web viewer also seem highly unlikely at this point.

Regarding GIS software, some commercial software can provide better support for semantic 3D city models, including the possibility to explore all of their semantics, and in particular FME (https://www.safe.com) has full support for CityGML and CityJSON from version 2020.0. However, this support is far from universal. Despite being the mostly widely used GIS program, ArcGIS currently has poor support for 3D city models. Admittedly, this is currently only known anecdotally, but more reliable testing of the capabilities of GIS software with respect to 3D city models is being conducted within the GeoBIM benchmark 2019 (Noardo, Biljecki, et al., 2019).

Despite the fact that some commercial GIS software might have good support for 3D city models, their price limits who can afford to use them and their closed nature limits how they can be used. For instance, it is not really possible to use such software to test variations and extensions of existing 3D city model standards, or to know exactly which invalid geometric configurations are handled (or not). In this sense, open-source GIS software is much more desirable, and a degree of support for CityJSON can be obtained in QGIS with the use of a plug-in (https://github.com/cityjson/cityjson-qgis-plugin). However, the relational (i.e., tabular) data model of QGIS and its associated attribute browser are not a good fit for the hierarchical data models of 3D city models.

Finally, many decent free 3D city model viewers are unfortunately closed source and Windows-only, including the FZKViewer (https://www.iai.kit.edu/1302.php) and the Elyx 3D Viewer (https://1spatial.com/products/elyx/

elyx-gis-platform/elyx-3d/), so they can only be used in other operating systems through virtual machines and compatibility layers such as Wine (https://www.winehq.org/), which comes with an increased use of resources, reduced performance and awkward workflows that clash with those in the remainder of the system.

Summarizing the related software presented above, we find that generic 3D viewers and 3D modeling software generally do not support 3D city model formats, and even when this support is added through plug-ins, they still lack the functionality to browse the hierarchies and complex semantics in 3D city models. A number of good web viewers for CityJSON do exist, but like all web-based software, their performance is limited compared to what is possible in desktop software. Some closed-source commercial GIS programs also have good support for 3D city model formats, but their price and closed nature limit who can use them and how they can be used. A few remaining 3D city model viewers are closed source and run only on Windows.

4 | SOFTWARE ARCHITECTURE AND PROCESSING METHODOLOGY

This section documents the methodology that is used by azul, including why particular algorithms, libraries and technical decisions were chosen. However, many of these aspects involve general city model processing techniques that should be useful for the any kind of software that needs to work with 3D city models, and particularly with CityJSON and CityGML.

4.1 | Software architecture and internal data model

The software architecture of azul mostly follows the standard model–view–controller design pattern, respectively corresponding to the `DataManager`, `MetalView` and `Controller` classes. The `DataManager` stores, processes and manages the data in azul's simple internal data model (Figure 2). The `MetalView` displays the 3D models, handles all graphic API calls and receives the input from user interactions. The `Controller` processes all input and sends commands to the `DataManager` and `MetalView`.

In addition, there are a few specialized classes that are called when necessary:

- `Enhanced_constrained_triangulation_2` (https://github.com/kenohori/cgal-etc) extends the constrained triangulations in CGAL (https://www.cgal.org) with functionality to store information in triangulation edges and to remove constraints. Note that the name of this class uses underscores to match the style used in CGAL.
- `PerformanceHelper` has some performance evaluation functionality (e.g., measuring elapsed times and memory used).
- `GMLParsingHelper` is used to parse CityGML. There is basic functionality to parse IndoorGML as well.
- `JSONParsingHelper` is used to parse CityJSON.
- `OBJParsingHelper` is used to parse OBJ.
- `POLYParsingHelper` is used to parse POLY.
- `OFFParsingHelper` is used to parse OFF.
- `TableCellView` defines a custom user interface element used to display elements in the azul sidebar with an icon and name.

Other non-class files are used to define Metal shaders, mathematical functions and wrappers to handle communication between different programming languages.

Since azul has been built to support multiple input formats, it has an internal data model (Figure 2), into which data are loaded by the various parsers described in the following section. Basically, the data model contains two

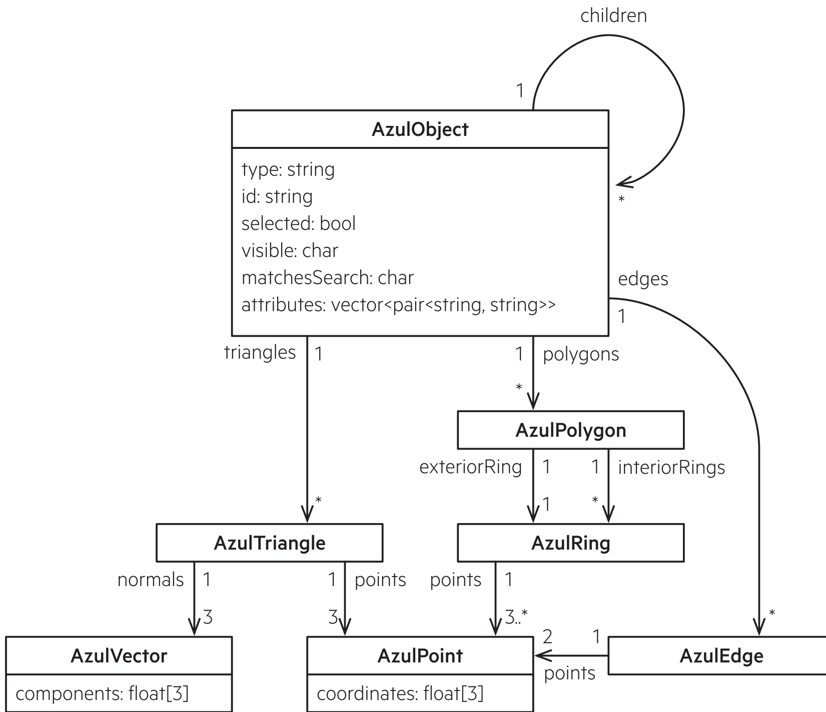


FIGURE 2 azul's internal data model

kinds of data: the semantics of a typical 3D city model, and the information required for azul's functionality. This is mainly evident in the `AzulObject`, which contains properties such as `type`, `id` and `attributes` to hold the city model's semantics, while also maintaining data such as the `selected` (yes/no), `visible` (yes/no/partially) and `matchesSearch` (yes/no/unknown) properties in order to represent the object's active state in the application.

The remainder of the data model describes the geometry of the model. Every `AzulObject` is composed of `AzulPolygon`, `AzulEdge` and `AzulTriangle` objects, of which only the polygons are loaded during the parsing process, while the rest are computed afterwards.

4.2 | Parsing and loading into the data model

4.2.1 | CityGML

Writing an efficient parser that accepts most CityGML files is a difficult task, mainly due to the overall complexity, verbosity and redundancy present in XML, GML and CityGML itself. Some of the problematic features include: the deep nesting of the hierarchy in even a small CityGML file (yielding large files that are hard to navigate), its poor match with the type systems of most programming languages (requiring custom serializers/deserializers), the multiplicity of ways in which things can be stored at many places in the hierarchy (requiring complex trees of decisions to provide full support), the use of XLinks (which means that a CityGML file is actually not a tree and requires expensive XPath-style queries to be traversed), and the ubiquity of errors in CityGML files (Biljecki et al., 2016).

In view of the above, azul generally follows a defensive strategy that tries to recover from common errors present in CityGML files. In other words, in order to make an attempt to load most data sets found in practice, azul does not behave as a validator. The basic XML parsing is done with the help of pugixml (https://pugixml.org),

which is among the fastest and most memory-efficient XML parsers available (https://pugixml.org/benchmark.html). It is a document object model (DOM) based parser (as the fastest XML parsers usually are), and our tests suggest that it can parse large CityGML files approximately 20 times faster than Cocoa's SAX-based XMLParser (the built-in Cocoa option used in many macOS applications) while using slightly less memory. Of relevance to CityGML, pugixml is also among the few efficient XML parsers that support XPath 1.0 queries (which can be used to resolve XLinks) and is thread-safe.

Since the XML namespace prefixes in CityGML are only recommendations and modules can be mapped multiple times to different prefixes (e.g., the CityGML core is often mapped to the default namespace and to the prefix `core`), azul ignores namespaces entirely by skipping to the character after the first colon (if any) in every XML tag. This simple method works because there are few if any name clashes in CityGML files, and it has the benefit of simplifying and speeding up the string matching operations used when parsing tags.

Since CityGML is a hierarchical format with various possible ways to encode things at several of the levels, azul also follows a hierarchical approach with branching conditions to parse CityGML files. Thus, specific methods handle different possible options for different levels in the hierarchy, and these methods call each other when necessary. The levels (slightly simplified) are summarized in Table 1 and explained in detail as follows:

1. At the *file* level, a CityGML model is detected by a tag `CityModel` with an attribute `xmlns` set to http://www.opengis.net/citygml/1.0 (CityGML 1.0) or http://www.opengis.net/citygml/2.0 (CityGML 2.0). After a model is detected, a fast index (a hash map using a `std::unordered_map`) of DOM nodes is built based on their GML `id`, which is a much faster option than performing an XPath query. All child nodes of the model are processed as potential CityGML objects.
2. At the *object* level, azul acts differently depending on a categorization of the XML tags:
   a. The tags representing semantically meaningful objects in the different CityGML modules (Figure 3) are loaded into the internal data model of azul. These are kept because they are either semantically meaningful by themselves, can contain meaningful attributes, or are necessary to distinguish between semantically different children of these nodes. The attributes of these types are parsed and loaded into the azul data model as well, and the children of their equivalent DOM nodes are parsed and processed as other potential CityGML **objects**.
   b. The tags that represent semantically meaningful objects *that provide only redundant information* are not included in the data model's hierarchy (i.e., they are flattened), but their child nodes are also parsed and processed as other potential CityGML *objects* (Figure 4). Examples include the `boundedBy` relation that contains an object's geometry (which can simply be associated with the object), and the `cityObjectMember`

TABLE 1 Summary of the CityGML parsing process

Level	Process
File	Detection of file type and construction of id index
Object	Based on XML tag of object:
	(a) if semantically meaningful, load into data model and parse children
	(b)–(c) if redundant, parse children but put at current level (i.e., flatten)
	(d) if a custom attribute, add in current object
	(e) if explicit geometry, process and add
	(f) if implicit geometry, obtain referenced geometry, transform, process and add
	(g) otherwise ignore
Polygon	Parse exterior and interior rings
Ring	Parse points (list or individually)

Relief
BreaklineRelief
MassPointRelief
RasterRelief
ReliefFeature
TINRelief

Building
Building
BuildingFurniture
BuildingInstallation
BuildingPart
IntBuildingInstallation
Room

Tunnel
HollowSpace
IntTunnelInstallation
RoofSurface
Tunnel
TunnelInstallation
TunnelFurniture
TunnelPart

Bridge
Bridge
BridgeConstructionElement
BridgeFurniture
BridgeInstallation
BridgePart
BridgeRoom
IntBridgeInstallation

WaterBody
WaterBody
WaterClosureSurface
WaterGroundSurface
WaterSurface

Transportation
AuxiliaryTrafficArea
Railway
Road
Square
Track
TrafficArea
TransportationComplex

Vegetation
PlantCover
SolitaryVegetationObject

CityFurniture
CityFurniture

CityObjectGroup
CityObjectGroup

LandUse
LandUse

GenericCityObject
GenericCityObject

Surfaces used in Building, Bridge and Tunnel
CeilingSurface
ClosureSurface
Door
FloorSurface
GroundSurface
InteriorWallSurface
RoofSurface
OuterCeilingSurface
OuterFloorSurface
WallSurface
Window

Multiple geometries
lodXFootPrint
lodXGeometry
lodXImplicitRepresentation
lodXMultiCurve
lodXMultiSolid
lodXMultiSurface
lodXNetwork
lodXTerrainIntersection
lodXRoofEdge
lodXSolid
lodXSurface

FIGURE 3 The semantically meaningful tags in azul cover types for semantic classes in the different CityGML modules (left three columns), classes with the same names in different modules (top right), and geometry types for different LoDs (bottom right). The X in the lodX... types refers to the many possible geometry types in different LoDs

Redundant semantic types	
auxiliaryTrafficArea	interiorHollowSpace
boundedBy	interiorRoom
breaklines	interiorTunnelInstallation
bridgeRoomInstallation	opening
cityObjectMember	outerBridgeConstruction
consistsOfBridgePart	outerBridgeInstallation
consistsOfBuildingPart	outerBuildingInstallation
consistsOfTunnelPart	outerTunnelInstallation
grid	reliefComponent
groupMember	reliefPoints
hollowSpaceInstallation	ridgeOrValleyLines
interiorBridgeInstallation	roomInstallation
interiorBridgeRoom	tin
interiorBuildingInstallation	trafficArea
interiorFurniture	

Geometry types that are not useful to show	
CompositeCurve	MultiSolid
CompositeSolid	MultiSurface
CompositeSurface	OrientableCurve
Curve	OrientableSurface
GeometricComplex	Shell
LineString	Solid
MultiCurve	Surface
MultiPoint	TIN
MultiGeometry	TriangulatedSurface

FIGURE 4 The tags for semantically meaningful but redundant objects (left) and those for geometric types that are generally not relevant for the end user (right)

relation for each object of a city model (which is obvious because such objects are under the city model in the hierarchy anyway). In this manner, CityGML files' nesting becomes much shallower and easier to navigate. In a similar case, the XML tags that represent geometric types that are generally not relevant for the end user (i.e., do not resolve ambiguities between LoDs or provide other meaningful information) are also flattened.

c. The tags that represent semantically meaningful objects that provide only redundant information, *but which can contain XLink-ed geometries* (Figure 5), are handled in a different case. azul has code to look for these linked geometries and add them to the hierarchy, but it is currently disabled (as of version 0.9) because it results in duplicate geometries being rendered, which causes flickering effects and makes it more difficult to select the appropriate objects when clicking. Similarly to the previous cases, the child nodes of these tags are also parsed and processed as other potential CityGML *objects*.
d. The tags corresponding to custom attributes (e.g., `stringAttribute` and `doubleAttribute`) are processed to add the attribute name and value (in the child node and grandchild node, respectively) to the current object.
e. The nodes corresponding to the tags for straightforward GML geometries (`Polygon`, `Rectangle` and `Triangle`) are passed along to the method that processes *polygons*.
f. The tags for `ImplicitGeometry` are handled separately by parsing the `transformationMatrix` and `referencePoint`. Then, azul parses the *objects* contained in the `relativeGMLGeometry`, either directly or by following the XLink in it (using the fast index of GML ids). Finally, it applies the transformation matrix and adds the reference point to all the points in the polygons obtained in the previous step.
g. azul ignores the tags for complex attributes (`address`, `externalReference`, `measureAttribute` and `genericAttributeSet`) for now because they cannot be neatly displayed in table form, the tags that would result in circular references in the hierarchy (`generalizesTo` and `parent`), and the tags that define geometries that end up covering the objects in the file (`extent` and `Envelope`).

3. At the *polygon* level, azul passes the nodes corresponding to the `exterior` and `interior` rings to the ring processing level. Note how much of the complexity inherent in parsing GML geometries is handled by flattening and traversing down from the higher-level nested tags (e.g., `LineString`, `Surface`, `Shell` and `Solid`).
4. At the *ring* level, azul processes lists of GML points, which can be in either `pos` (individual points) or `posList` form (lists of coordinates).

4.2.2 | CityJSON

JSON is much easier to parse than XML—often requiring no external libraries—and thus parsing CityJSON is relatively straightforward. Early versions of azul thus had very simple CityJSON pre-1.0 parsing code based on Niels Lohmann's user-friendly JSON for Modern C++ library (https://github.com/nlohmann/json), which, despite not being among the fastest JSON parsing libraries (https://github.com/miloyip/nativejson-benchmark#parsing-time), already competed with the highly optimized CityGML parser.

However, with the release of CityJSON 1.0, an effort was made to optimize the CityJSON parser in azul to the same level as the CityGML parser. For this, azul 0.9 has a new parser based on the highly optimized experimental simdjson library (https://github.com/lemire/simdjson) (Langdale & Lemire, 2019), which uses modern processors' SIMD instructions to speed up parsing. It is worth noting that despite spending less time developing azul's CityJSON parser than the CityGML parser, azul is now able to parse CityJSON files twice or three times faster than the same files in CityGML.

Overall, the code in azul needed to parse CityJSON is a much simplified version of the CityGML code. Since the hierarchical structure in CityJSON does not contain redundant elements, it is not necessary to remove them when loading it into the internal data model.

The code thus has hierarchical methods to parse CityJSON city objects, geometry objects (i.e., the array under city objects' "`geometry`"), the "`boundaries`" nested array containing a geometry object's polygons, individual polygons, and individual rings. However, instead of using recursion to handle the unknown nesting level in

CityGML files, the CityJSON parsing code follows a predictable order, keeping only a few iterators that are used to access the relevant JSON objects (e.g., the top-level elements) during a breadth-first traversal of a file's structure.

There are, however, a few parts that are somewhat more difficult to deal with than the CityGML code. These are: the need to check an indirect reference to the "`vertices`" array to get the coordinates of a vertex based on its index; the parallel traversal of an object's geometry and semantics; and the transformation of a vertex's coordinates using the transform object.

4.2.3 | IndoorGML

In the interest of supporting the development of indoor models, azul has basic support for IndoorGML as well. This support is based on the CityGML code described above, detecting instances of `CellSpace` to put in azul's data model and parsing GML geometries using the same shared code as for CityGML. While azul's IndoorGML support has not been tested extensively, it is able to load the few IndoorGML models that are available at this point (http://indoorgml.net/resources/). Since it is not really possible to test the IndoorGML functionality further at this point, more development on this front will be done once more models are available.

4.2.4 | Other formats

Finally, azul has basic support for OBJ and OFF files, which are often used during the development of 3D city models, and are widely supported and exported by most 3D modeling and geometric processing software. There is also basic support for POLY files, which can be used in Triangle (Shewchuk, 1996) and TetGen (Si, 2015), and is therefore useful for triangulated or tetrahedralized building models. Since these formats are all quite simple, azul's support has been built mostly from scratch based on plain C-style string comparisons.

4.3 | Triangulation of polygons

Computing the triangles for every polygon requires a more complex process, mainly because of the need to obtain a triangulation of the polygons, which may be invalid (e.g., degenerate, self-intersecting or having badly nested rings) or significantly non-planar. In order to obtain a plane that can be used as a base for the triangulation of a polygon, azul computes the best-fitting plane passing through its vertices using the least-squares function in the CGAL (https://www.cgal.org) Principal Component Analysis package (https://doc.cgal.org/latest/Manual/packages.html). While this procedure is admittedly computationally expensive, simpler methods that were used before (e.g., using the plane passing through the first three non-collinear points or using Newell's method to compute the normal of the plane) do not obtain a good fit in all cases.

After this plane is obtained, azul follows an improved version of the process detailed in Ledoux, Arroyo Ohori, and Meijers (2012). Briefly, all of a polygon's rings are closed if necessary, and its edges are projected to the plane and added as constraints to a constrained Delaunay triangulation (Shewchuk, 1996) that follows an odd–even rule. In this triangulation, only the edges that have been added an *odd* number of times (as parts of the line segments defining constraints in the original input) remain constrained, while those that are added an *even* number of times become unconstrained (and are possibly retriangulated to satisfy the constrained Delaunay criterion). This is done using a custom class that extends the triangulations in the CGAL Triangulations package. However, the original 3D coordinates of each vertex are kept as an attribute in the 2D vertices on the plane, so that they can be reused after the polygon triangulation has been computed.

Afterwards, the triangles in the triangulation are labeled as belonging to the interior or the exterior of the polygon, also using an odd–even rule. The triangles that are reachable by passing through an odd number of constrained edges from the exterior of the polygon—which can be obtained from the "infinite" face of the triangulation—are labeled as interior triangles, while those reachable by passing through an even number of times are labeled as exterior triangles.

Finally, the interior triangles are put into the triangle storage of the corresponding object. If a vertex already existed in the original polygon, its original 3D coordinates (which were stored) are reused in order to obtain a better fit with the original model. However, when a vertex is new (e.g., lying at an intersection of two edges of the original polygon), its 3D coordinates are obtained by assuming that it lies exactly on the computed plane.

Once the triangles and edges of a data set's polygons have been created, the original polygons are cleared to free up memory.

4.4 | Loading and displaying data from the data model

At this point, the triangles to be rendered are split into many objects, most of which contain only a few triangles. Since rendering them in this form is rather inefficient, azul first generates much larger single-color triangle buffers that can be directly loaded to graphics memory and displayed. These are created by aggregating the triangles of all objects up to a user-definable size, which is currently set to 16 MB. At the same time as these buffers are being created, the coordinates of the triangles are translated and scaled to fit within [−1.0, 1.0] on the *x*-, *y*- and *z*-axes.

Since the viewer performs a perspective projection, and the default parameters put the camera at (0.0, 0.0, −1.0) and pointing toward the origin with a field of view of 60 degrees, scaling the data ensures that all the files loaded are visible, properly centered and roughly fit in the viewer's window (depending on its height–width ratio). In addition, this avoids accuracy issues due to the floating precision of the original data (e.g. mainly when the coordinates are in a reference system that uses high values; Goldberg, 1991).

A simplified version of the process for triangles is followed for edges, which are usually rendered in black, as well as for a set of edges defining the bounding box of the data. Special buffers are created for selected objects, which are rendered with yellow triangles and red edges to make them easily distinguishable from the remainder of the objects.

After being generated, these buffers are loaded into graphics memory and rendered by the viewer using the previously defined colors and Gouraud shading (i.e., the final color being decided by adding ambient, diffuse and specular colors based on the triangle normals).

For the computation of the final 2D coordinates on the screen, azul uses the model, view and projection matrices that are typically used in graphics applications. The model matrix is defined as a multiplication of a matrix to translate the model to the center of rotation, a matrix to rotate the model, and a matrix to shift it back to its original location. Translation and scaling are not applied here because it is more intuitive to apply translations along the data plane, and scaling is best done by changing the field of view of the camera. The view matrix applies a transformation to look from the camera location to the data center. The projection matrix applies a perspective projection with the defined field of view, and a proper aspect ratio based on the height and width of the viewer window.

In addition to the 3D viewer section of the window (Figure 6), azul contains a sidebar with three sections: (a) a search panel; (b) an outline view that shows the visible objects; and (c) a table view to browse the selected object's attributes. The search panel activates an incremental (i.e., real-time) search of the typed text within the loaded objects' id, type or attributes. The outline view is a selectable list of objects with expandable items to show their hierarchy, individual toggles for visibility, icons, types and ids. The table view lists the attributes (names and values) one by one.

Redundant geometric types with potential xlinks	
baseSurface	patches
curveMember(s)	pointMember(s)
element	segments
exterior	solidMember(s)
geometryMember	surfaceMember(s)
interior	trianglePatches

FIGURE 5 The tags for semantically meaningful but redundant objects that can nevertheless contain XLinked geometries

FIGURE 6 Opening a 3D city model of Leiden, Netherlands

4.5 | Interaction model

The interaction model of azul has been designed to work as native Mac (and iOS) applications usually do, using scrolling for panning, the pinch gesture for zooming, and the rotation gesture to rotate.

4.5.1 | Panning

Scrolling (along any angle) on a trackpad or mouse pans the model. Since 3D city models usually extend horizontally much more than they do vertically (e.g., kilometers versus hundreds of meters), it is possible to define an approximate data plane around which the data set lies. In azul, this is defined as the plane that passes orthogonal to the z-axis at the midpoint between the minimum z and maximum z coordinates of the data set.

azul thus pans along this data plane, which helps to keep the data set at the same distance during the process. In order to perform this operation, azul first obtains the two-dimensional scrolling motion as Δx and Δy

coordinates from the Cocoa API, which is multiplied by a scrolling sensitivity factor that is proportional to the field of view of the camera. This keeps the panning motion consistent regardless of the zoom level. This motion is converted from camera coordinates to object coordinates (by multiplying it by the inverse of the model-view matrix) and applied.

Afterwards, azul applies a correction to ensure that the motion maintains the distance to the data plane. For this, it computes a similar transformation to the previous one that offsets the change in the distance to the data plane, which is computed at the center point of the window.

4.5.2 | Zooming

Zooming is activated using the pinch gesture on trackpads, or alternatively by right-clicking and dragging (on any input device). For the pinch gesture, the Cocoa API already returns a difference in the magnification factor that can be applied to the tangent of the field of view (because the field of view is an angle rather than a length). For the dragging operation, azul first calculates its own magnification factor by multiplying the Δy of the motion by a zoom sensitivity parameter, and then it applies the same process.

4.5.3 | Rotation

Rotation is activated using the rotation gesture on trackpads, or alternatively by left-clicking and dragging (on any input device). Similarly to the pinch gesture, the Cocoa API returns a rotation that can be directly applied around the rotation axis, which corresponds to the z-axis in camera coordinates.

For the dragging operation, azul implements Arcball rotation, which is an intuitive method that is equivalent to dragging a point on a sphere that is centered on the window and directly behind the screen (i.e., touching it from behind in the center of the window). In order to compute this motion, azul first calculates the distance to the sphere for the last and current positions of the pointer. These are considered as the z coordinates of the two positions, which are used together with the x and y coordinates to compute the angle between the two points as the inverse of the cosine of their dot product. This is the angle of rotation, which is applied around an axis orthogonal to the two vectors given by the two points. To easily obtain a vector along this axis, it is sufficient to obtain the cross-product of the last and the current cursor positions.

4.5.4 | Selection

Selection can be performed by clicking an object in the 3D view, which selects the frontmost object at the top level of the hierarchy in the data model. Alternatively, it is possible to select an object in the sidebar with the mouse or keyboard. Both methods allow for multiple selections or deselections by pressing the command key (for individual selections or deselections) or the shift key (for ranges of objects). The selection in the sidebar is simple since the sidebar items keep pointers to the corresponding objects in the data model.

In order to select a clicked object in the 3D view, azul computes the location of the clicked (x, y) location on the near and far planes, which in projection coordinates are simply given by $(x, y, \pm 1.0)$, and can be converted back to object coordinates by multiplying them by the inverse of the model–view–projection matrix. azul then does a brute force computation to find the closest triangle (among all objects) that is hit by a ray going from the point on the near plane to the far plane. The object with that triangle is then selected or deselected.

4.5.5 | Centering view

azul implements a double-click method to center the view at the clicked location in the 3D view. Alternatively, it is possible to double-click on an object in the sidebar to center the view at that object.

For the method on the 3D view, azul computes a ray passing through the clicked location on the near and far planes. Afterwards, this ray is intersected with the data plane to obtain the location at which the view should be centered. A translation to reach this location is computed and applied, and an offset to keep the data plane at the same distance is computed and applied as well.

For the sidebar method, azul computes the centroid of the vertices of the selected object and centers the view using the same steps.

5 | IMPLEMENTATION DETAILS

The vast majority of azul (about 96.6%, according to GitHub) is written in C/C++, including the parts responsible for reading and parsing the input files, triangulating every polygonal face, making GPU-friendly buffers for all triangles and edges, and storing the geometry and attributes of the 3D city models in the in-memory data model. This corresponds to the `DataManager` and most of the auxiliary classes. By using C++, it is possible to obtain good performance while taking advantage of various useful libraries (e.g., pugixml, simdjson, boost and CGAL), and the cross-platform compatibility of C++ also makes it easier to port azul to other operating systems in the future.

This core C++ code is then tied to smaller pieces of Swift code (about 2.3% of the total), which allows azul to use native macOS APIs at the lowest level to perform fast graphics processing with the Metal API and matrix operations with the Apple's simd library, and at the highest level to provide native user interface and mouse/keyboard/trackpad controls with the Cocoa API. These parts of the code could be ported to other operating systems using appropriate APIs (e.g., .NET for Windows and Qt for Linux, plus something like Vulkan for the graphics calls).

For the remainder of the code, since Swift and C++ cannot communicate directly, data is passed using plain C data types and (dumb) pointers through an Objective-C++ double wrapper (0.7% of the code) with a pure Objective-C header (0.3%). An example of this process is shown in Figure 7.

In order to make string comparisons as fast as possible, most string comparisons are done using plain C functions like `strcmp` and `memcmp`. However, numbers are parsed using Boost's (https://www.boost.org) Spirit X3's `float _` parser, which is significantly faster than the standard `std::atof` and `std::strtod` functions (https://stackoverflow.com/questions/5678932/fastest-way-to-read-numerical-values-from-text-file-in-c-double-in-this-case)—about three times than the implementation in LLVM's libc++ library (https://libcxx.llvm.org).

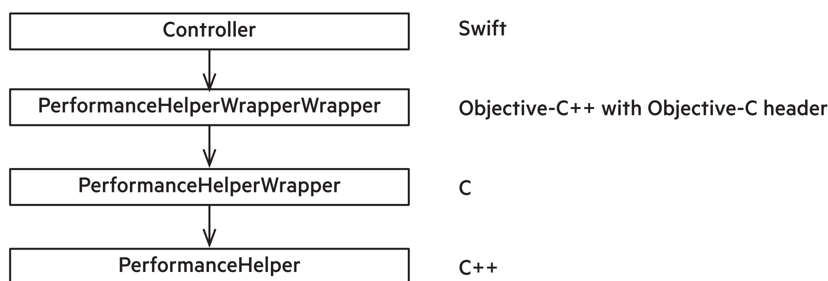
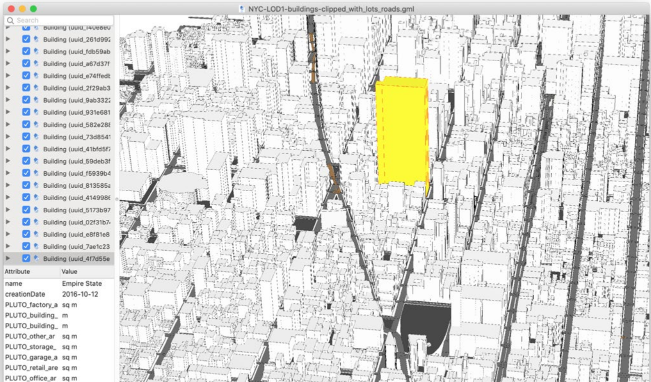
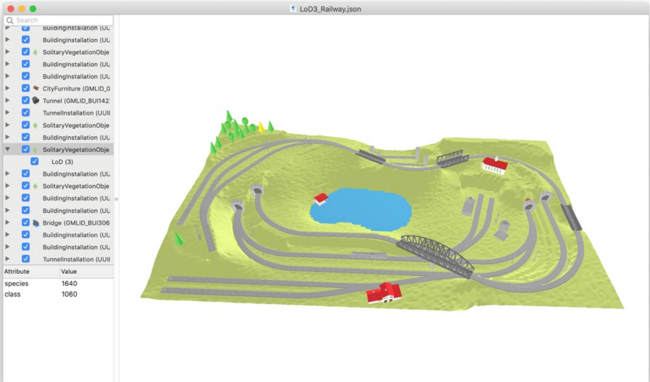


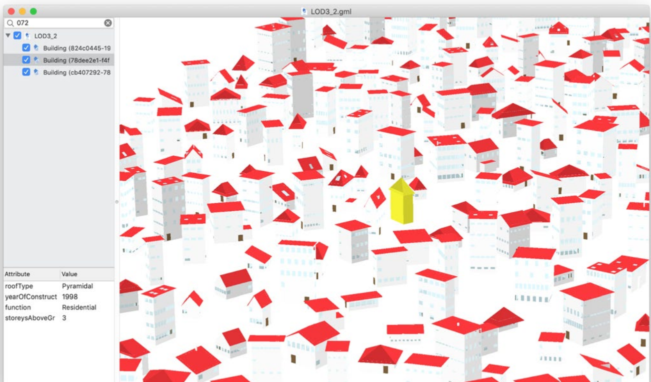
FIGURE 7 A double wrapper of the `PerformanceHelper` class used to communicate between Swift and C++. This is possible because Swift is compatible with Objective-C, and so it can call methods defined in the Objective-C header of the double wrapper class, which in its Objective-C++ implementation can then call methods defined in the C++ class



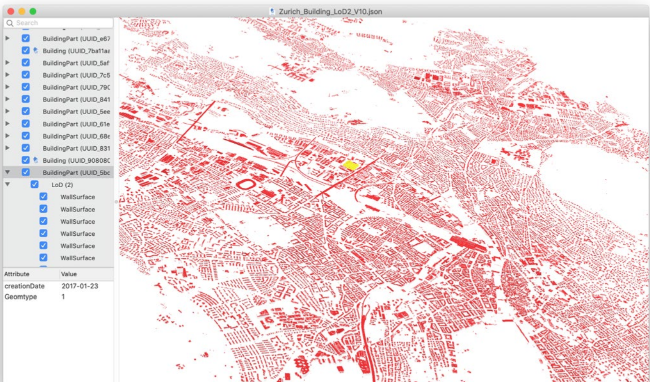
(a)



(b)



(c)



(d)

FIGURE 8 A few of the data sets that have been tested with azul. (a) New York City http://www.gis.bgu.tum.de/en/projects/new-york-city-3d/. (b) CityGML 2.0 sample dataset http://web.archive.org/web/20161129082447/http://www.citygml.org/index.php?id=1539. (c) Dataset generated with Random3Dcity https://github.com/tudelft3d/Random3Dcity. (d) Zürich https://data.stadt-zuerich.ch/dataset?q=&tags=3d-stadtmodell)

TABLE 2 Performance comparison of open data sets that are available in CityGML and CityJSON

Data set	Format	Size	Max. memory	Time
The Hague	CityGML	24.2 MB	191.8 MB	1.79 s
	CityJSON	3.2 MB	138.2 MB	0.41 s
Montreal	CityGML	59.8 MB	264.6 MB	1.28 s
	CityJSON	6.3 MB	160.0 MB	0.84 s
New York	CityGML	620.8 MB	2.53 GB	24.23 s
	CityJSON	117.5 MB	1.41 GB	15.41 s
Railway	CityGML	47.2 MB	247.1 MB	0.98 s
	CityJSON	5.3 MB	179.1 MB	0.65 s
Rotterdam	CityGML	16.8 MB	156.2 MB	0.48 s
	CityJSON	3.2 MB	131.8 MB	0.29 s
Vienna	CityGML	38.8 MB	266.7 MB	3.75 s
	CityJSON	6.4 MB	206.2 MB	1.23 s
Zurich	CityGML	3.04 GB	7.74 GB	169.19 s
	CityJSON	302 MB	3.08 GB	37.22 s

6 | AVAILABILITY AND RESULTS

azul's source code is freely available under the GPL 3 license, which satisfies the requirements of the libraries that azul depends on: Boost (Boost Software License), CGAL (LGPL 3 and GPL 3), GMP (LGPL 3 and GPL 2), MPFR (LGPL 3), pugixml (MIT), and simdjson (Apache). It is also available in binary form either from its GitHub repository (https://github.com/tudelft3d/azul) or from the Mac App Store (https://itunes.apple.com/app/azul/id1173239678?mt=12). Issues can be submitted through GitHub (https://github.com/tudelft3d/azul/issues).

azul has been extensively tested with a wide variety of open data sets in CityJSON and CityGML, including all of those included in the list of cities with open 3D city models from the 3D geoinformation group at Delft University of Technology (https://3d.bk.tudelft.nl/opendata/opencities/) (Figure 8), as well as with various non-open data sets used in other research projects. It is able to load correctly almost every valid data set tested so far, although further work is needed to properly display data sets with different units along different axes (e.g., lat-long and EPSG:4979) or those with very large coordinates (where operations with oating-point issues can become a problem; Goldberg, 1991).

Many invalid data sets also work in azul, as the software is able to deal well with most common geometric issues, including open rings, invalid polygons, and non-planar faces, as well as with many syntactic and semantic errors, such as broken XLinks and repeated ids. The known data sets that cannot be open in azul have major problems that cannot be easily recovered from, such as badly malformed XML and XML tags that do not follow the hierarchy prescribed in the CityGML standard.

In order to test the performance of azul, experiments with open data sets that are available in both CityGML and CityJSON were conducted using a MacBook Pro (Retina, 15-inch, Mid 2015) with a 2.8 GHz quad-core Intel i7 processor and 16 GB of RAM. The results, which are summarized in Table 2, show close to linear behavior, although performance radically decreases when virtual memory is needed.

7 | CONCLUSIONS AND DISCUSSION

3D geoinformation has been produced and amply discussed for decades and at this point ought to be considered as an established technology. The availability of open-source 3D GIS software is, however, still rather poor and clearly

lags behind closed alternatives in many areas. In a revealing example, QGIS (as the main open-source GIS) only got basic 3D visualization support in early 2018 and good support toward the end of that year. Reading and working with 3D GIS formats using only open-source tools is still often difficult and limiting, and the panorama on the building information modeling side—dominated by very complex and/or proprietary formats—is significantly worse.

Admittedly, azul represents only a small contribution toward the creation of more open-source 3D GIS software, but it does so in a key research and application area with few good existing alternatives. It also fulfills the design goals we established at the beginning of the project (regarding file formats, speed, smoothly displaying large areas, robust methods, and intuitive interaction). While further work is needed for full cross-platform compatibility, the vast majority of the code is already built in C++ using cross-platform libraries.

Apart from the software itself, its methodology (or relevant parts of it) can be replicated in other programs to improve their processing and visualization of 3D city models. We think that the following aspects are particularly interesting: (a) using generic data models that are suitable for multiple formats; (b) following a well-documented robust approach that tries to recover from errors in data sets; (c) flattening complex hierarchies when possible; (d) using state-of-the-art parsers; and (e) developing an interaction model that is suited to 3D city models and the hardware that is available.

Moreover, we hope that azul can serve as a good example of open-source software developed in academia (with an open process in GitHub and responding to issues and suggestions), and that the ambitious approach taken with it (with optimized code, native APIs and a polished user interface) can inspire other research groups and software developers to release their source code and to improve their applications.

Admittedly, developing open-source software in academia is still rather challenging on several fronts: software is often not considered as valuable research output (by both funding agencies and internal evaluations in academic institutions), spending time on improvements and maintenance does not fit in typical research processes, and attempting to have papers published based on software involves challenging many preconceptions (e.g., accusations of “not doing real science”). The creator of SageMath (https://www.sagemath.org) puts it in a pithier form: “Every great open-source math library is built on the ashes of someone's academic career (https://www.youtube.com/watch?v=6eIoYMB_0Xc).”

However, it is worth noting that many research trends are currently chipping away at these challenges, including the movements toward open science and replicability, as well as the explosion in data-based science and the increased use of statistical/computational methods. In our particular case, having a supportive research group has been the key factor making software development possible. While this might seem like a incurring a cost with no gain, showing a track record of software development has allowed us to attract new projects, students and researchers.

A lot of open-source GIS software relies on cross-platform GUI libraries (e.g., Qt), but this one-size-fits-all approach often results in a poor experience across all platforms. Using a cross-platform C++ core with native APIs for the user interface is a feasible alternative that results in software that can follow a platform's design language and behaves according to user expectations.

As of this writing, all of the main desired features for azul have been implemented and released in azul 0.9. Since this release includes the new CityGML and CityJSON parsers, some issues are expected, which will be collected and fixed for version 1.0 later in 2020. In addition, this release will also add options for unloading files, icons for the missing semantic classes, and the possibility to add/change the colors used for different semantic classes. Object hierarchies will also be reordered to better show XLinks and parent–child relationships between objects. In the longer term, some other desired features include better handling of big coordinates, improved incremental search with viewing of matching objects, adding support for materials and textures, shifting the data plane upwards/downwards, and better keyboard navigation.

ACKNOWLEDGMENTS

I would like to thank Stelios Vitalis for his insightful remarks that helped improve this article. This work was supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 677312 UMnD).

ORCID

Ken Arroyo Ohori https://orcid.org/0000-0002-9863-0152

ENDNOTE

1 See https://3d.bk.tudelft.nl/opendata/opencities/ for a full list.

REFERENCES

Bartie, P., Reitsma, F., Kingham, S., & Mills, S. (2010). Advancing visibility modelling algorithms for urban environments. *Computers, Environment & Urban Systems*, *34*(6), 518–531.

Biljecki, F., Ledoux, H., Du, X., Stoter, J., Soon, K. H., & Khoo, V. H. S. (2016). The most common geometric and semantic errors in CityGML datasets. *ISPRS Annals of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, *4*(2/W1), 13–22.

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., & Çöltekin, A. (2015). Applications of 3D city models: State of the art review. *ISPRS International Journal of Geo-Information*, *4*(4), 2842–2889.

Cignoni, P., Callieri, M., Corsini, M., Dellepiane, M., Ganovelli, F., & Ranzuglia, G. (2008). MeshLab: An open-source mesh processing tool. In V. Scarano, R. De Chiara, & U. Erra (Eds.), *Eurographics Italian Chapter Conference* (pp. 1–8). Aire-la-Ville, Switzerland: Eurographics Association.

Diakité, A., & Stoter, J. (2017). *Eindrapport scoping studie voor integratie GeoTOP en BIM: Als input voor de ontwikkeling van basis registratie ondergrond* (Technical report). Delft, Netherlands: Delft University of Technology.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, *23*(1), 5–48.

Jakubiec, J. A., & Reinhart, C. F. (2013). A method for predicting city-wide electricity gains from photovoltaic panels based on LiDAR and GIS data combined with hourly Daysim simulations. *Solar Energy*, *93*, 127–143.

Langdale, G., & Lemire, D. (2019). Parsing gigabytes of JSON per second. *VLDB Journal*, *28*, 941–960.

Ledoux, H. (2018). val3dity: Validation of 3D GIS primitives according to the international standards. *Open Geospatial Data, Software & Standards*, *3*, 1.

Ledoux, H., Arroyo Ohori, K., 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 & Standards*, *4*, 4.

Ledoux, H., Arroyo Ohori, K., & Meijers, M. (2012). Automatically repairing invalid polygons with a constrained triangulation. In J. Gensel, D. Josselin, & D. Vandenbroucke (Eds.), *Multidisciplinary research on geographical information in Europe and beyond: Proceedings of the 15th AGILE International Conference on Geographic Information Science*, Avignon, France (pp. 13–18). Avignon, France: AGILE.

Liang, J., Gong, J., Li, W., & Ibrahim, A. N. (2014). A visualization-oriented 3D method for efficient computation of urban solar radiation based on 3D-2D surface mapping. *International Journal of Geographical Information Science*, *28*(4), 780–798.

Nguyen, H. T., & Pearce, J. M. (2012). Incorporating shading losses in solar photovoltaic potential assessment at the municipal scale. *Solar Energy*, *86*(5), 1245–1260.

Noardo, F., Biljecki, F., Agugiaro, G., Arroyo Ohori, K., Ellul, C., Harrie, L., & Stoter, J. (2019). GeoBIM benchmark 2019: Intermediate results. *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, *42*(4), 47–52.

Noardo, F., Ellul, C., Harrie, L., Overland, I., Shariat, M., Arroyo Ohori, K., & Stoter, J. (2019). Opportunities and challenges for GeoBIM in Europe: Developing a building permits use-case to raise awareness and examine technical interoperability challenges. *Journal of Spatial Science*, *65*(2), 209–233.

OGC. (2004). *OpenGIS Geography Markup Language (GML) Implementation Specification (Version 3.1.1)*. Wayland, MA: Open Geospatial Consortium.

OGC. (2008). *OpenGIS City Geography Markup Language (CityGML) Encoding Standard (Version 1.0.0)*. Wayland, MA: Open Geospatial Consortium.

OGC. (2012). *OGC City Geography Markup Language (CityGML) Encoding Standard (Version 2.0.0)*. Wayland, MA: Open Geospatial Consortium.

Peters, R., Ledoux, H., & Biljecki, F. (2015). Visibility analysis in a point cloud based on the Medial Axis Transform. In F. Biljecki & V. Tourre (Eds.), *Proceedings of the 2015 Eurographics Workshop on Urban Data Modelling and Visualisation*, Delft, Netherlands (pp. 7–13). Goslar, Germany: Eurographics Association.

Shewchuk, J. R. (1996). Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In M. C. Lin & D. Manocha (Eds.), *Applied computational geometry: Towards geometric engineering* (Lecture Notes in Computer Science, Vol. *1148*, pp. 203–222). Berlin, Germany: Springer-Verlag.

Si, H. (2015). TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software*, *41*(2), 11.

How to cite this article: Arroyo Ohori K. azul: A fast and efficient 3D city model viewer for macOS. *Transactions in GIS*. 2020;24:1165–1184. https://doi.org/10.1111/tgis.12673