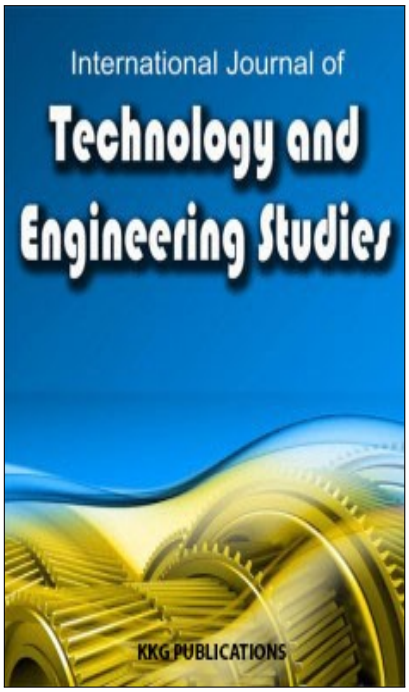

Key Knowledge Generation
Publication details, including instructions for author and subscription information:
http://kkgpublications.com/technology/

A Web Serverless Architecture for Buildings Modeling

ENRICO MARINO [1], DANILO SALVATI [2], FEDERICO SPINI [3], CHRISTIAN VADALA [4]

[1, 3] Department of Engineering, Roma Tre University, Rome, Italy
[2, 4] Department of Mathematics and Physics, Roma Tre University, Rome, Italy

Published online: 22 June 2017

To cite this article: E. Marino, D. Salvati, F. Spini and C. Vadala, "A web serverless architecture for buildings modeling," *International Journal of Technology and Engineering Studies,* vol. 3, no. 3, pp. 93-100, 2017.
DOI: https://dx.doi.org/10.20469/ijtes.3.40001-3

To link to this article: http://kkgpublications.com/wp-content/uploads/2017/3/IJTES-40001-3.pdf

KKG PUBLICATIONS

International Journal of Technology and Engineering Studies
vol. 3, no. 3, pp. 93-100, 2017

IJTES

A WEB SERVERLESS ARCHITECTURE FOR BUILDINGS MODELING

ENRICO MARINO [1]*, DANILO SALVATI [2], FEDERICO SPINI [3], CHRISTIAN VADALA [4]

[1, 3] Department of Engineering, Roma Tre University, Rome, Italy

[2, 4] Department of Mathematics and Physics, Roma Tre University, Rome, Italy

Keywords:
Modeling
Serverless Architecture
User Collaboration

Received: 12 February 2017
Accepted: 10 April 2017
Published: 22 June 2017

Abstract. This research aims to develop a Web-based buildings' modeling tool which overcomes the performance and development difficulties relying on a unidirectional data flow design pattern and a serverless architecture, respectively. This paper introduces an effective Web architecture for buildings' modeling that leverages the serverless pattern to dominate the developing complexity. The resulting front-end application, powered by Web Components and based on unidirectional data flow pattern, is extremely customizable and extendible by means of the definition of plugins to augment the UI or the application functionalities. As regards the modeling approach, it offers (a) to model the building drawing the 2D plans and to navigate the building in a 3D first-person point of view; (b) to collaborate in real-time, allowing to work simultaneously on different layers of the project; (c) to define and use new building elements, that are furniture or architectural components (such as stairs, roofs, etc.), augmenting a ready-to-use catalog. This work suggests a path for the next-coming BIM online services, matching the BIM approach's professional collaboration requirements typical of the BIM approach with the platform that supports them the most: the Web.

INTRODUCTION

Nowadays we are seeing a relentless migration of software products toward services accessible via the Web medium. This is mainly due to the undeniable benefits in terms of accessibility, usability, maintainability and spreadability granted by the Web medium itself. Nevertheless these benefits don't come without a cost: performance and development complexity become major concerns in the Web environment.

In particular, due to the introduction of several abstraction layers it is not always feasible to "port" a desktop application into the Web realm, an aspect to be taken into account even for the relevant hardware differences among all the devices equipped with a Web Browser. It can be even more arduous to tackle the inherent distributed software architecture (a client/server one at least) induced by the Web platform. Nevertheless increasingly rich and complex Web applications began to appear, supported by the enriched HTML5 APIs, which thanks to the WebGL [1] (which enables direct access to GPU), Canvas [2] (2D raster APIs) and SVG [3] (vectorial drawing APIs), have paved the way for the entrance of Web Graphic Applications.

In this work we report about our endeavor toward the definition of a Web-based buildings' modeling tool which overcomes the aforementioned performance and development difficulties relying on a unidirectional data flow design pattern and on a serverless architecture, respectively.

A serverless architecture, on the contrary of what the name may suggest, actually employs many different specific servers, whose operation and maintenance don't burden the project developer(s). These several servers can be seen as third party services (typically cloud-based) or functions executed into ephemeral containers (may only last for one invocation) to manage the internal state and server-side logic. Real-time interaction among users jointly working on the same modeling project, is for example achieved via a third party APIs for remote users' collaboration.

The tool user interface, entirely based on web components pattern, has been kept as simple as possible: the user is required to interact mainly with two-dimensional symbolic placeholders representing parts of the building, thus avoiding complex 3D interactions. The modeling complexity is thus moved from the modeler to the developer which fills out an extendible catalog of customizable building elements. The modeler has only to select the required element, place and parametrize it according to the requirements. It is obvious that a large number of building elements has to be provided to ensure the fulfillment of the most modeling requirements. The remainder of this document is organized as follows:

Section 2 provides an overview of related work. Section 3 reports about the application user experience. Section 4 presents adopted architectural solutions. Finally, Section 5 contains some conclusive remarks.

*Corresponding author: Enrico Marino
†Email: salvati.danilo@gmail.com

RELATED WORK

In this section we highlight some remarkable experiences aligned with the aim of our project. There are plenty of desktop applications worth to be mentioned and analyzed, but in the following we deliberately focus on Web-based works.

Shapespark offers a web viewer of remarkable quality that allows the user to move inside a virtual 3D indoor environment. Modeling phase is served in the form of plugins for different Desktop proprietary solutions.

Playcanvas is a complete and powerful web-based game creation platform which offers an integrated physical engine and a whole set of functionalities to support modeling. Although powerful and relatively simple to use, it doesn't focus on buildings' modeling. Floorplan has been developed by Autodesk specifically for the architectural field, and for indoor renewal projects in particular. It is a 2D modeling tool which offers also a 3D walk-through mode.

[4] introduced a Web modeling and baking service for indoor environments. The modeling tools expose a 3D interaction the user may not be accustomed to, a hitch we tried to outflank by avoiding 3D modeling interaction and let the user only face a "metaphoric" 2D interface. As regards support for users' collaboration it is worth to be mentioned the Operational Transformation (OT) approach [5]: a group of nodes exchanges messages without a central control point. Two main properties hold in this setup: (i) changes are relative to other user's changes (it works on "diffs") and (ii) no matter in which order concurrent changes are applied, the final document is the same. In our serverless architecture however, external (third parties) central synchronization points are allowed, making complexity introduced by protocols like OT less effective.

APPLICATION EXPERIENCE

The application experience focuses on supporting the user in a building modeling task. The exploited modeling approach requires the user to face as much as possible a two-dimensional interface which allows her to define the plan and to place complex architectural elements (here called building elements) on it. Such building elements can be found in a pre-filled catalog, and when required can be further configured and customized through a side panel. This modeling approach moves part of the complexity toward the developer of the customizable building elements, leaving to the final user the task to place and to configure the employed elements. A rich catalog of elements is thus crucial to answer to the users' modeling requirements.

Once the floor-plan has been defined according to the place-and-configure approach, the system can automatically generate a 3D model which can be explored externally or in first person view, as shown in Figure 1f. Each building element in fact comprises of either a 2D generating function (2Dgf) or a 3D generating function (3Dgf), used to obtain models used in the 2D floorplan definition and in 3D generated model respectively. The tool also has support for layers the user can exploit to organize her project, for example to group together semantically homogenous elements.

Building Elements

Along with the aforementioned 2D and 3D generating functions, an element is fully specified by its univocal name and its properties, used by the user for customization. Each building element inherits from its prototype (one and only one). The prototype maps both the inherent characteristics and user interactions needed to add the element to the project and/or configure it.

The catalog comprises then of four different types of elements:

Lines. An element which belongs to this category is drawn selecting a start point and an end point. To move it one can drag one of the end-points or can drag the entire line. An example: a wall.

Openings. An opening is an element that is linked to a line-element, making a "hole" on it. The user creates a new opening by dragging an opening-element on a chosen line-element. Examples are doors and windows.

Areas. An area is an element which may be generated by defining the boundary vertices. A room basement is a good example of an area-element, which is automatically generated from walls (that are line-elements).

The algorithm for the basement computation follows these phases:

(i) search of biconnected component by means of Hopcroft-Tarjan algorithm (see [6]);

(ii) removal of edges that are not part of a biconnected component;

(iii) search of all cycles through an algorithm that does a double check of each edge sorted by angle;

(iv) search of maximal cycles correspondent to perimeter edges by an application of Gauss's area formula;

(v) removal of maximal cycles.

Object. An element that is freely inserted into space with a drag and drop interaction. Examples are tables and chairs.

User Interface

Figure 1a shows the application's user interface. It consists of the following components:

Content-area: It displays the main content and it is chosen by the user. Developers can extend the available content-

areas and so the functionalities of the system using techniques described in Section 4. Those which were already implemented by us are: (i) building elements catalog viewer (figure 1b), (ii) 2D drawing area (figure 1d), (iii) 3D viewer (figures 1c and 1e), (iv) 3D first person viewer (figure 1f).

Toolbar: It contains buttons mapping all operations the user can do. Type and number of buttons change according to the content-area chosen.

Sidebar: It contains the list of building elements added to the current layer and the list of layers of the model. Moreover it permits to customize properties' values for the selected elements.

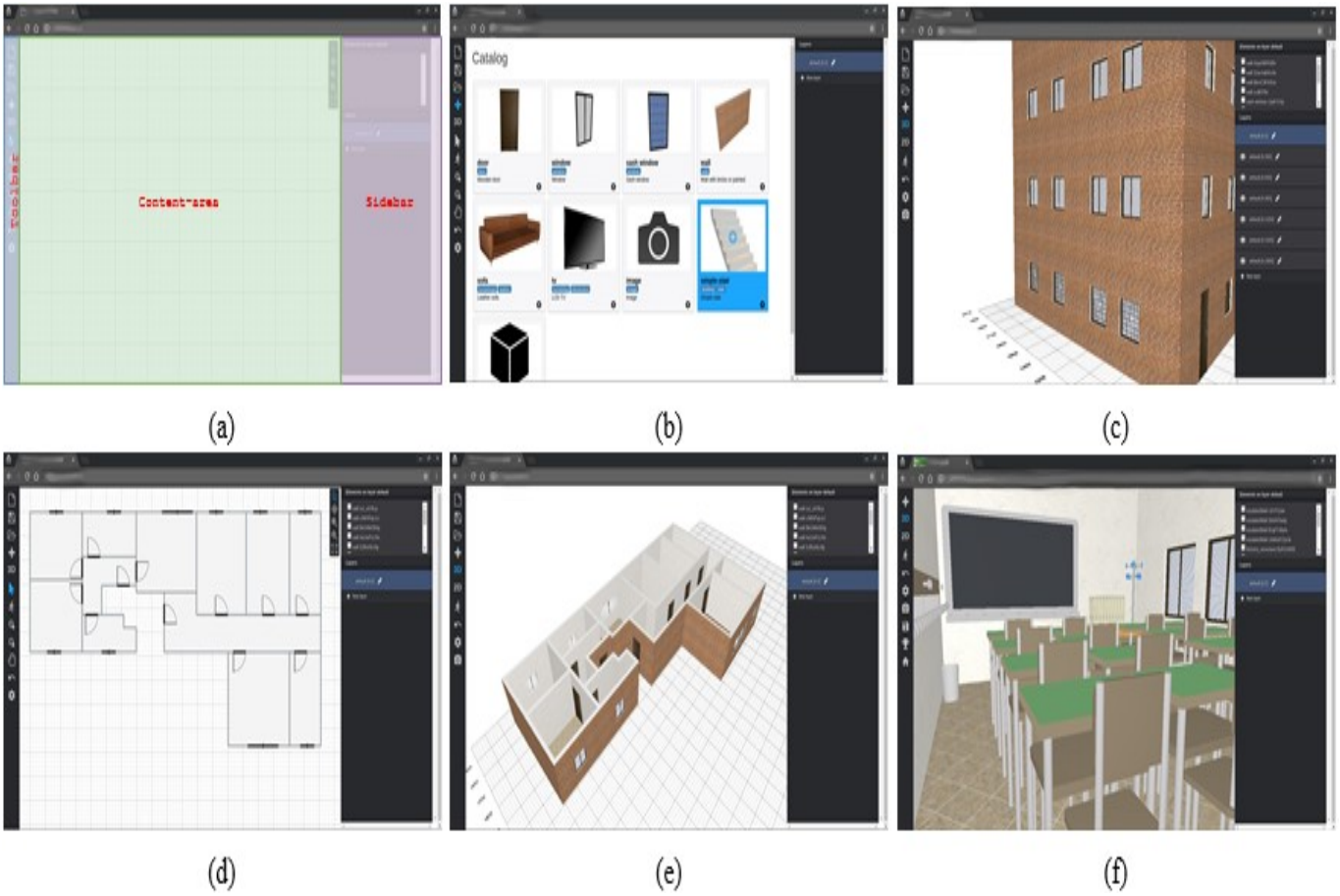

Fig. 1. User Interface: (a) main interaction areas; (b) building elements catalog; (c) interaction with a building; (d) 2D interaction with a floor plan; (e) 3D interaction from external point-of-view (f) 3D interaction from a first-person point-of-view

Serverless Architecture

According to [7], the deployed serverless architecture makes extensive use of third party services, or "as-a- Service-like" components, that replace ad-hoc software and hardware and fulfill the same tasks.

In particular we have been able to delegate the following aspects to external and specialized services: application distribution, user management, model generation (heavy computation), user collaboration and state storage.

Distribution. To serve the application resources (JavaScript files, images, etc.) we rely on a KeyCDN (a Content Delivery Network) and avoid any kind of webserver. This implies high availability and performance, but also represents a centralized method to upgrade client application without any user explicit action: once files have been updated on the CDN, after the subsequent reload of the web application, any user is provided with the updated version of the application.

User Management. User management activities, which requires a backend running code to accomplish user subscription and authentication, are delegated to Auth05, a Backend-as-a-Service (BaaS) specialized on accounting.

Model Generation. The generation of 3D models may require a while. In that case, especially on low-performing hardware, the computation can't be performed on the client-side,

but rather on a powerful server which generates the model on demand, taking as input the parametrization made by the user through the web interface.

On this purpose we chose to move this computation on AWS Lambda, a Function-as-a-Service (FaaS) on which runs the Python code responsible to generate 3D models, that automatically scale on the basis of the number of model generation tasks.

Users' Collaboration. As regard the users' collaboration, it requires some sort of synchronization among them. To use a single point of synchronization results in a much simpler architecture, so instead of relying on complex peer-to- peer architecture, which however would be a good choice aligned with the serverless paradigm we are pursuing, we opted for a synchronization mechanism based on Firebase, a Backend-as-a-Service working as a coordination manager.

It broadcasts each change made by a user to all the others. Conflicts are avoided exploiting layers: each user locks the layer she is working on.

State Storage. The whole state of a modeling project is represented as a JSON document. To save and reload project state the user can easily download the corresponding JSON document, or, to remain "in the Cloud", the same document can be stored on a DataBase-as-a-Service (DBaaS), Orchestrate for example.

In this case we only rely on endpoints to save and to load a document: on a save event, client application serializes the state and passes it to the DBaaS which stores it, ready to serve it on a load request issued at a later time. Once the client receives back the document, the state is automatically restored thanks to its reactive architecture.

The real support for serverless architecture is here not provided by the service that actually stores the document, which as stated can be replaced even with a file download, but more precisely, by the software architecture that supports serialization and a reloading of the state. This architecture addresses two main concerns:

(i) centralized immutable state and

(ii) a reactive UI (i.e. modifications of the state reflect automatically on the user interface).

Centralized Application State

The application state is modeled using the data structure shown in Listing 1. It is essentially a collection of layers, each containing a collection of vertices, lines, areas and objects, each one of which is captured in a structure composed by:

(i) information required by the object prototype;

(ii) references mapping the relationship to other objects;

(iii) metadata, namely the object customization entry point. Listings 2 and 3 give examples of data structures adopted to model a vertex and a line, respectively. Information redundancies are exploited to decrease access times. Collections of objects are indexed by id thus allowing lookup in constant time, The selected field of each layer, grants direct access to selected elements without searching. The state can be loaded one layer at a time to support state fragmentation thus allowing to deal with very big building modeling project.

Unidirectional data flow. The described data structure represents the centralized state required by the unidirectional data flow pattern [8] exploited by the application via Redux.js library.

The pattern prescribes that the state may be modified only by specific actors, called reducers, whose activities are triggered by specific actions which contain all the information needed by each reducer to accomplish the state change. Each application feature has to be implemented therefore as a couple of well-isolated pieces of code (action/reducer).

Preliminary experiments on 2D drawing tools, in fact, highlighted the development complexity of an application of this kind in terms of large internal state modified by several user interactions, which was to be listened to and applied, resulting in a high coupling level between application logic and user interface. In our setup instead we defined a state engine, which represents the application logic, comprising of actions and reducers, and encapsulates the centralized state. On this layer can transparently rely different interfaces.

```
{
  "width": 3000, // canvas width
  "height": 2000, // canvas height
  "unit": "cm", // unit of measurement
  "selectedLayer": "layer-1", // current layer
  "layers": {
    "layer-1": {
      "name": "default",
      "id": "layer-1",
      "altitude": 0,
      "opacity": 1,
      "visible": true,
      "vertices": {
        "HJAe59YF8Ux": {...}
        // ...
      },
      "lines": {
        "Hype99FK88x": {...}
        // ...
      },
      "openings": {
        "r1jaKYUIg": {...}
        // ...
      },
      "areas": {
        "Byg1oFKUIe": {...}
        // ...
      },
      "objects": {
        "rkKU89U8e": {...}
        // ...
      },
     //selected element
      "selected": {
        "vertices": [],
        "lines": [],
        "openings": [],
        "areas": [],
        "objects": ["rkKU89U8e"]
      }
    }
  }
}
```

Fig. 2. JSON serialized state, overall structure

```
{
  "id": "HJAe59YF8Ux",
  "x": 201,
  "y": 891,
  "prototype": "vertices",
  "selected": false,
  "lines": ["Hype99FK88x", "S1w-hqKKL8e"],
  "areas": ["Byg1oFKUIe"]
}
```

Fig. 3. JSON serialized state, vertex structure

Immutability pattern. Immutability pattern [9] is also applied, to avoid side effects on state changes performed by reducers. The state can be seen as an immutable tree structure whose changes are applied as follows: (i) clone the previous state s obtaining a new state s'; (ii) apply changes on the cloned state s'; (iii) Update reference from s to s'.

```
{
  "id": "Hype99FK88x",
  "type": "linear",
  "prototype": "lines",
  "vertices": ["HJAe59YF8Ux", "r11Z59tKIUg"],
  "openings": ["r1jaKYUIg", "BJVZ2M9FIIx"],
  "selected": false,
  "properties": {
    "height": 300,
    "thickness": 20,
    "coverA": "bricks",
    "coverB": "bricks"
 }
```

Fig. 4. JSON serialized state, line structure

It is worth noting that this approach provides out-of-the-box support for undo/redo operations: an older/newer state can be restored by means of a replacement of the current state with the previous/next one.

Despite its simplicity, this pattern can nevertheless lead to memory waste, due to the several copies of the state that must be held in memory. We addressed this issue using Immutable.js, a library which exploits structural sharing via hash maps tries and vector tries, thus minimizing the need to copy or cache data.

User's interaction. We modeled the user's interaction as a Finite State Machine (FSM) where each node corresponds to a mode (i.e. to a possible application state), and each edge corresponds to an action (e.g. a user's interaction) that can be set off in the current state, (typically a JavaScript event mapping a user's interaction such as click or mousemove).

Figure 5 shows the FSM relative to the wall drawing interaction. The three nodes correspond to three modes: (i) node idle, the waiting mode of the application where no action has been taken yet; (ii) node waiting_drawing_wall, where the user has selected wall design tool but he hasn't started the draw phase yet; (iii) Node drawing_wall, where the user has placed the starting point of the wall.

Reactive Component Based UI

The UI has been developed following the Web Components pattern [10], supported by React.js framework. The main idea is to define the frontend application as a collection of independent components, each one referencing a specific subset of the centralized state and able to render itself according to the actual values of that portion of the state. Web Components spawn from for high level generic containers, like the toolbar or the catalog, to very fine grained ones, buttons for example. The most interesting are the viewers of the building model: the 2D-viewer and the 3D-viewer.

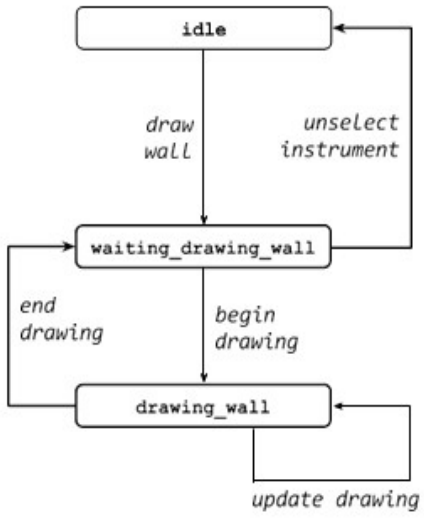

Fig. 5. Subgraph of the state machine that shows a wall creation

Viewers. A viewer is a pivotal component since it shows the building model and allows user's interaction with it. We built a 2D-viewer and a 3D-viewer.

The 2D-viewer invokes the 2Dgf of the building elements added to the model and renders its output using SVG elements. To cope with frequent updates coming from the user's drawing interaction, it exploits the Virtual DOM [11], which permits to update only the modified part thus avoiding complete redrawing of the scene. To perform pan and zoom operations, typically necessary in this kind of tool, we develop an ad-hoc React component named ReactSVGPan- Zoom. The 3D-viewer invokes the 3Dgf of the building elements added to the model and renders its output using WebGL primitives via Three.js. It has been implemented a diff and patch system, standardized in [12]: Three.js objects are associated with building elements inside the state, so every time the user triggers an action that results in a state alteration, the application computes the difference between the old state and the new one and changes only the affected object. In particular we can have the following operations: (i) add, (ii) replace and (iii) remove.

Figure 3 shows the interaction among viewers, state-engine and catalog. A viewer reacts to any state change updating its internal state and displaying the changes applied to the model. The generating functions are pulled from the catalog where a descriptor for each building element can be found.

CONCLUSION AND RECOMMENDATIONS

In this work we outlined a serverless architecture to support buildings' modeling in a Web environment. The serverless architecture that gives benefits in terms of availability, reliability, scalability, easiness of deployment, maintainability and upgradability is obtained by implementing the application logic as a client-side only centralized state Web application exploiting the unidirectional data flow pattern. This approach allows for an easy-to-serialize state (in the form of a JSON document) that can be pushed on a third party document oriented DB-as-a-Service and loaded back in the frontend reactive architecture, which transparently reloads the state once its serialized version is passed in. The application itself is served by a CDN thus avoiding any need for web server. Offline routines rely on Function-as-a-Service platform as well as users management and collaboration features.

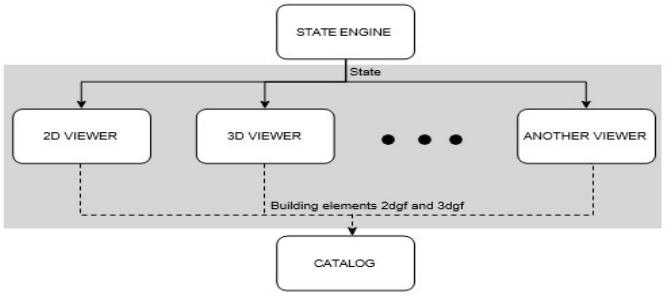

Fig. 6. Viewers' architectural scheme

Services costs. As regards costs to be paid for third party services, we have estimated an expense of less than 100$ per month for about 5000 users, for the steady state operation. Currently however we have not exceeded the free tier offered by each one of the exploited services.

Metior project. The described architecture has been successfully employed by the authors as foundation for the Metior project [13], a tool to support selective deconstruction of buildings in the pursuit of a "zero waste" model.

As an outcome of this experience we will be able to gather some usability tests, useful to further improve the users' experience.

Future developments. At this stage of development each single layer of the state is required to fit in memory. Although this shortcoming can be easily circumvented by slitting up a big layer, we are currently addressing a practical way to scale also inter-layer, by allowing a selective loading of the layer content.

Declaration of Conflicting Interests

No competing interests are declared by the authors.

Acknowledgments

Authors would like to thank GEOWEB S.p.A., a web service company owned by Sogei S.p.A. and CNGeGL Italian National Board of Quantity Surveyors, for supporting this work. Thanks are extended to Stefano Perrone for developing the models shown in the Figures 1d, 1e, and 1f.

REFERENCES

[1] D. Jackson. (2014). *WebGL specification* [Online]. Available: https://goo.gl/ln7SNw

[2] J. Munro, J. Mann, I. Hickson, T. Wiltzius and R. Cabanier. (2015). *HTML canvas 2D context* [Online]. Available: https://goo.gl/ut02Cw

[3] P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers and J. Watt. (2011). *Scalable Vector Graphics (SVG) 1.1.* [Online]. Available: https://goo.gl/kj61ix

[4] F. Spini, E. Marino, M. D. Antimi, E. Carra and A. Paoluzzi, "Web 3D indoor authoring and VR exploration via texture baking service," in *proceedings of the 21st International Conference on Web3D Technology,* Anaheim, CA, pp. 151-154, July 22-24, 2016.

[5] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *ACM SIGMOD International Conference on Management of Data,* Portland, OR, pp. 399-407, 1989.

[6] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Communications of the ACM,* vol. 16, no. 6, pp. 372-378, 1973.

[7] M. Roberts. (2016). *Serverless architectures* [Online]. Available: https://goo.gl/K9ezLc

[8] T. Hos. (2016). *Reactivity, state and an unidirectional data flow* [Online]. Available: https://goo.gl/qUTrlg

[9] J. Long. (2015). *Immutable data structures and JavaScript* [Online]. Available: https://goo.gl/THHovV

[10] A. Russell. (n.d). *Web components and model driven views* [Online]. Available: https://goo.gl/LjmckY

[11] J. Rotolo. (2015). *The virtual DOM vs the DOM* [Online]. Available: https://goo.gl/UBb70G

[12] P. Bryan and M. Nottingham. (2013). *JavaScript object notation (JSON) patch* [Online]. Available: https://goo.gl/fl8y6e

[13] E. Marino, F. Spini, A. Paoluzzi, D. Salvati, C. Vadala, A. Bottaro and M. Vicentino, "Modeling semantics for building deconstruction," in *proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications,* Porto, Portugal, pp. 274-281, Feb. 27-March 01, 2017.

— This article does not have any appendix. —