Optimizing 3D Object Visualization on the Web

João Victor de Figueiredo Leite[1],
João Marcelo Xavier Natário Teixeira[1,2(✉)], and Veronica Teichrieb[1]

[1] Voxar Labs, CIn, Universidade Federal de Pernambuco, Recife, Brazil
{jvfl, vt}@cin.ufpe.br
[2] DEINFO, Universidade Federal Rural de Pernambuco, Recife, Brazil
jmxnt@cin.ufrpe.br

Abstract. With the rise of new technologies for visualizing 3D information in the browser, a trend can be observed concerning the growing use of such technologies in web-based applications, due to browsers being present in virtually every device. Also, a growth can be observed in the 3D printing field, since the printers are becoming cheaper as the technology evolves. This study aims to develop a loader and a web visualizer for the 3MF format, and test its performance across desktop and mobile devices, searching for an optimized way of displaying 3D printing data in browsers. To test the validity of the loader, a 3D printing simulator was also implemented and tested across platforms. It was discovered that 3MF is better than STL for visualizing 3D content on the web, due to its greater capabilities, extensibility, and even a smaller loading time given the right optimizations.

Keywords: 3D visualization · 3MF · Three.js · Web

1 Introduction

Lately, new web technologies in the field of 3D graphics have been developed, such as WebGL [1], "Three.js" [10] and Babylon [4], which are enabling a pervasive cross-browser form of processing and displaying graphical data without most of the portability concerns traditional technologies have, leaving only typical cross-browser API differences as a possible problem. These new technologies have already enabled powerful new tools to be developed [23], and their performance can be fine-tuned to work well both on desktop and mobile platforms [24].

Currently, those visualization technologies are being explored widely and a myriad of examples are available. Many of them are now related to a field that has been gaining attention in the last years: 3D printing.

3D printing, or additive manufacturing, is the process of transforming information contained in a digital three-dimensional object into a physical object. It achieves such feat by extracting horizontal cross-sections of the virtual objects, which are then printed and laid down until the whole object is created [12].

Throughout the last couple of years, 3D printing technology has risen in popularity, even further than what was expected according to experts [2]. This gave spawn to a

S. Lackey and R. Shumaker (Eds.): VAMR 2016, LNCS 9740, pp. 287–298, 2016.
DOI: 10.1007/978-3-319-39907-2_27

series of very differentiated applications ranging from manufacturing composite elements [17] to printing biomaterials [15].

However, the file format considered as the de facto standard, STL (Standard Triangulation Language) [19, 21, 22], is falling behind since it is limited in its representational capabilities in regard to the constantly evolving technological scenario. Therefore, new initiatives were founded in order to find a suitable format that can be extended as needed, providing support to emerging technologies, such as AMF (Additive Manufacturing Format) [20] and 3MF (3D Manufacturing Format) [13].

2 ThreeMF Loader

The loader proposed in this work was implemented through the use of the "Three.js" library to parse 3MF files in accordance with the specifications [3]. This section discusses the "Three.js" library and the loader's architecture.

According to its specification [3], a 3MF file is in fact a zip file that obeys the Open Packaging Conventions [18]. For this work, the main files observed inside the zip are the 3Dmodel.model, which stores 3D models as triangular meshes, and the image textures files, because both contain the core of what is necessary to fully display the 3D model that the 3MF file represents. It is noteworthy that the 3Dmodel.model file is, in fact, an XML file.

2.1 Three.Js

"Three.js" is a lightweight 3D library that simplifies 3D graphics usage inside a browser [10]. It features a simple API and has different options of rendering engines, with the default engines being: HTML5 canvas and SVG elements, CSS3D and WebGL. For the purposes of this work, the WebGL renderer was used together with the HTML5 canvas element, so the models could be rendered to the screen. The following classes were utilized in this work to fully represent a 3MF model:

- **Scene:** Manages the rendering of several models on the screen. Each model must be constructed and added to the scene separately by using one of the provided classes in "Three.js".
- **Geometry:** Contains all the information about the objects vertices, faces, colors and texture UVs, and also includes positional information.
- **Face3:** Represents a triangle shaped face in space by referencing vertices previously added to a Geometry. As an added bonus, this class also stores which color or texture used.
- **Mesh:** Related to Geometry and the Material which will be used to display it.
- **MeshPhongMaterial:** The material utilized for this work, defining that the Phong shader will be used to represent the objects after they are loaded.

By utilizing such classes, one can fully describe a triangular mesh, regardless of size, even if such mesh contains more complex information such as color and textures. For a

concrete example about the library's initialization and object construction, please refer to the source code of this project [11], and also to the examples at the "Three.js" website.

2.2 Architecture

For the definition of the loader's architecture, current loaders at the "Three.js" base code were studied, and a general structure was identified. As a result, a new class called ThreeMFLoader was created as per the needs of the structure.

The primary function of "Three.js" loaders is the load function, which receives a url and returns a model. But, despite having a similar signature, some loaders differed about the function return type, which was defined either as Mesh or Geometry objects depending on the loader. Considering that the function of a loader should not affect how the loaded model is displayed, it was decided that the ThreeMFLoader should just return a Geometry and a list of the accompanying textures, leaving the materials choice to the final user.

Since the 3MF is a zip file, libraries regarding the use of zip files in Javascript had to be found, because no other "Three.js" loader had to deal with a zipped file at the time this work was done. So, the "zip.js" library [14] was chosen as a means to do so, and functions were defined to load the "3Dmodel.model" file and the related texture image files from the 3MF zip file.

However, the "zip.js" library uses Web Workers, which are essentially an implementation of threads in the Javascript environment, implying in the files being extracted from the zip file in an asynchronous manner. A simple synchronization control was then devised through counting all the files that needed to be extracted, and then subtracting such counter by one each time a file was fully extracted.

With all the files available for parsing after extracting, new functions were defined to parse the "3Dmodel.model" file. Given that such file is an XML file, the default Javascript's DOM parser was utilized to navigate through it, with several new functions being created to modularize each of the XML file nodes concerns.

The root function for XML processing, called processXML, is responsible for separating and storing the information of tags that can later be referenced by other elements, such as the texture2d, colorgroup and texture2dgroup tags.

Besides the processXML function, the parseObjects function can be defined as being fundamental in the parsing process. It extracts all the vertex and triangle tags from an object node and inserts them onto a Geometry as Face3 objects. Each triangle node can also hold a reference to either a colorgroup or a texture2dgroup, and thanks to the pre-processing done by the processXML function, such information is available to be used and inserted in the Face3 object, or inside the Geometry object, as it happens in the UV mappings case.

Most of the functions besides the processXML and parseObjects can be classified as helper functions, since they only take care of simpler details, like extracting the coordinates of a vertex. In such way, they are not discussed here, but are available for consultation at the source code of the project [11].

3 Printing Simulator

In order to validate the loader usefulness in a real web application, a basic 3D printing simulator was implemented. This section will present the basic implementation details of the Simulator, describing the libraries.

3.1 Implementation

After the model is in the appropriate structure, to simulate the additive manufacturing process one has to divide it into layers, effectively slicing it. To achieve slicing, a CSG (Constructive Solid Geometry) library was utilized, namely the "CSG.js" library [6].

The "CSG.js" library implements CSG Boolean operations by using BSP (Binary Space Partitioning) trees, but at the time of implementation of this work it was severely outdated [6]. A suitable updated version was found inside the source code of the OpenJSCad project [8], which enabled the implementation of the slicing feature through the use of CSG's intersection operation.

Despite being an updated version, the "CSG.js" basic data structures differ from the ones found in "Three.js", and as such a third component was needed in order to enable interoperability between the libraries; the "ThreeCSG.js" [5] was used to do so. However, the new component was also outdated. Thus, the "ThreeCSG.js" library had to be updated, which presented a need to better understand the "CSG.js" data structures.

Similarly to "Three.js", "CSG.js" has a set of classes designed to represent 3D models, from which some were selected based on their likeness to "Three.js" data structures. Those are the CSG, Polygon, Vertex and Vector3D classes, described as follows:

- **CSG:** Represents a 3D model and contains a list of Polygons. Its chosen counterpart in the "Three.js" library is the Geometry class.
- **Polygon:** A 3D polygon represented by a list of vertices. Since it describes a face in the high level CSG object, this class was used to be the equivalent of a Face3 object, despite being more general.
- **Vertex:** Represents a single 3D point and can be directly correlated to a "Three.js" vertex.
- **Vector3D:** Require for the construction of a CSG Vertex. It represents a vector or a point in 3D space.

After updating the "ThreeCSG.js" library, the objects could be easily con- verted between CSG and Geometry objects, giving "Three.js" full access to the "CSG.js" capabilities. The intersection operation was available to "Three.js" Geometry objects, and a parallelepiped could be defined in a way that slices varying heights by intersecting it with an arbitrary Geometry, given that the parallelepiped has depth and width equal to the geometry (Fig. 1).

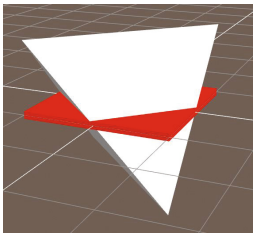

Fig. 1. The defined parallelepiped intersecting a tetrahedron.

As result, two functions were defined in a separate file called "3DPrinterSimulator.js":

- **generateSlices:** Receives a mesh and a slice height, and then produces the sliced mesh as Geometry objects.
- **displaySlices:** Receives the "Three.js" scene, a list of mesh slices, a delay time and feedback function. It renders each slice in the scene according to the delay set by the user, then executes the feedback function so the user may know that the simulation has ended.

By using such functions, one can effectively simulate the 3D printing process on a "Three.js" geometry by first generating the slices with the desired height, and then displaying them (Fig. 2).

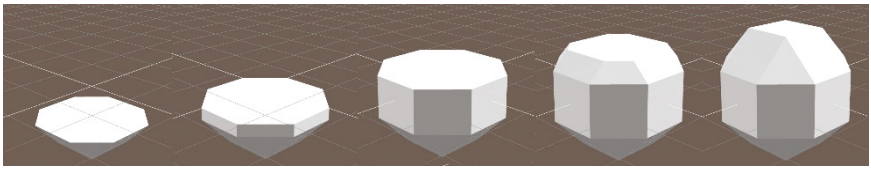

Fig. 2. Intermediary simulation results of the 3D printing process.

4 Results

To confirm the pervasiveness and effectiveness of the technologies utilized, the loading process was tested both on mobile and desktop platforms. This section describes the tests performed in this work, which were executed in four models: Tetrahedron, Rhombicuboctahedron, Dodecahedron chain and Heartgears (Fig. 3).

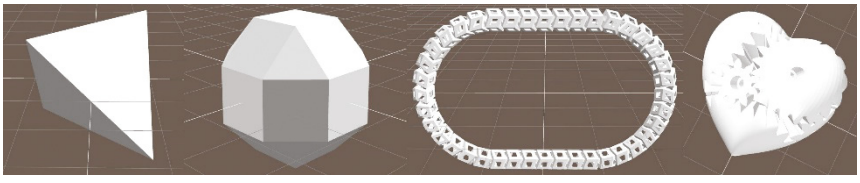

Fig. 3. Loaded 3MF models.

4.1 File Format Comparison

Although the STL format has been the *de facto* standard for printing over the past two decades, it is severely limited and fail to address the evolving needs for 3D printing [19, 21, 22]. Some extensions have been proposed, but they were not widely accepted [16, 25]. As a side-effect of the lack of extensions, nowadays the STL file format is only able to save and load simple 3D meshes based on triangles, and has no support for texture, color, and materials. It is important to notice that an STL file can be represented in either ASCII or Binary, with the only difference between them being their file size in bytes, since the Binary format is considerably smaller than the plain ASCII file.

As for the 3MF file format, it proposes to fix some of the STL problems, offering a specification with flexible rules for extensions. In such way, although it still only supports triangular models, it can be easily extended in order to support more complex representations. Additionally, texture, color, and materials are already supported out of the box [3]. It also comes with only one representation, (i.e., a zip file).

An in-depth comparison between file sizes is shown as follows (Table 1), where files containing exactly the same information were compared in order to ascertain which is better suited for web visualization given the file size taken to represent the same object.

Table 1. File format sizes (in KB).

Model	Number of Triangles	STL Size	Binary STL Size	3MF Size
Tetrahedron	4	2	1	2
Rhombicuboctahedron	44	12	3	2
Dodecahedron chain	7680	1323	376	62
Heartgears	30636	6363	1502	326

Thus, it can be observed that the size of a 3MF file grows slower than STL files, even the binary one. Also, since the smaller file size makes downloading faster, the 3MF file is better suited for web visualization in regard to file size.

4.2 Loading Time

The file size only affects the download speed of the model; its effect can be nullified if parsing takes too long. Hence, tests gauging was conducted to learn whether the unzipping and parsing of the 3MF file will negate the file size benefits. The tables in sequence (Tables 2, 3, 4 and 5) describe the loading times of the STL and 3MF formats. In the tests, three types of devices were used: PC, smart phone, and tablet. Each model was loaded 100 times on each device, and the average times and their standard deviations were recorded.

The devices specifications are listed as follows:

PC ASUS N56JR-EH71
- **Processor** 2.4GHz Intel I7 4700HQ
- **Video Card** NVIDIA Geforce GTX 760M with 2GB of memory
- **RAM** 12GB
- **Browser** Chrome 47
- **OS** Windows 8.1

Phone iPhone 5S
- **Processor** 1.3GHz Apple A7
- **Video Card PowerVR G6430 RAM 1GB**
- **Browser Safari 9**
- **OS iOS 9.2**

Tablet Samsung Galaxy Note 2014 Edition
- **Processor** 1.3GHz quad-core Exynos 5420
- **Video Card** Mali-T628
- **RAM** 3GB
- **Browser** Android Native Browser
- **OS** Android 4.3

Table 2. STL file parsing time (average time in ms).

Model	PC (Standard Deviation)	Phone (Standard Deviation)	Tablet (Standard Deviation)
Tetrahedron	0.185 (0.164)	1.022 (1.468)	0.994 (0.958)
Rhombicuboctahedron	1.085 (0.336)	4.435 (2.268)	11.816 (3.130)
Dodecahedron chain	239.523 (29.469)	581.258 (56.211)	856.718 (102.663)
Heartgears	1185.450 (106.550)	5879.429 (187.686)	4614.742 (1177.846)

Table 3. Binary STL file parsing time (average time in ms).

Model	PC (Standard Deviation)	Phone (Standard Deviation)	Tablet (Standard Deviation)
Tetrahedron	0.085 (0.136)	0.185 (0.398)	0.371 (1.219)
Rhombicuboctahedron	0.090 (0.047)	0.128 (0.068)	0.711 (0.307)
Dodecahedron chain	7.329 (2.486)	9.244 (2.034)	33.484 (17.117)
Heartgears	36.959 (7.898)	47.643 (4.248)	122.044 (12.729)

Table 4. 3MF unzipping times (average time in ms).

Model	PC (Standard Deviation)	Phone (Standard Deviation)	Tablet (Standard Deviation)
Tetrahedron	133.605 (15.447)	61.501 (29.877)	140.397 (12.603)
Rhombicuboctahedron	134.651 (50.706)	62.273 (27.945)	148.852 (13.751)
Dodecahedron chain	199.252 (104.746)	242.116 (43.144)	806.121 (190.775)
Heartgears	313.421 (62.440)	716.328 (221.484)	2482.954 (98.507)

Table 5. 3MF parsing times (average time in ms).

Model	PC (Standard Deviation)	Phone (Standard Deviation)	Tablet (Standard Deviation)
Tetrahedron	0.171 (0.137)	0.579 (0.166)	2.413 (4.560)
Rhombicuboctahedron	0.423 (0.175)	1.314 (1.075)	4.070 (2.546)
Dodecahedron chain	42.404 (17.966)	56.067 (12.799)	438.604 (165.474)
Heartgears	308.822 (73.671)	392.871 (725.502)	2230.832 (161.264)

The binary STL file format seems more advantageous in loading times, especially when the 3MF unzipping time is factored in. Nonetheless, it is important to notice that the unzipping time can still be compensated by the smaller file size on larger models, since unzipping files has a relatively short time and is not prone to variations given the users access to a fast Internet connection.

4.3 Loaders Comparison

During the development of this work, one of the objectives was to contribute to the "Three.js" source code by the addition of a 3MF loader. However, a third party loader was submitted before, and a comparison between the one previously submitted (current Three.js' 3MF loader) [10] and the one developed here was drawn in regards to capabilities and loading times.

As far as capabilities go, the loader present at the "Three.js" code parses some minor tags from 3MF that the loader proposed here does not, such as the metadata tag, which contains information such as the file author, creation date, among others. As for the loader here proposed, it provides support to textures and colors as extra features.

Architecture-wise, it is important to notice that the libraries utilized to unzip files were different. While the proposed loader uses the "zip.js" library, the "Three.js" loader uses the "jszip.js" library [7]. When tested in regards to unzipping time, "jszip.js" proved to be much faster than "zip.js" (Table 6). However, it did not work on the tested phone and tablet, even when changing browsers. The browsers tested were Safari, Chrome, Opera, Firefox and the Native Android Browser.

Table 6. Time taken by each loader to unzip files in PC (average time in ms).

Model	"Three.js loader PC Average (Standard Deviation)	Our loader PC Average (Standard Deviation)
Tetrahedron	0.208 (0.364)	133.605 (15.447)
Rhombicuboctahedron	0.277 (0.906)	134.651 (50.706)
Dodecahedron chain	5.189 (2.071)	199.252 (104.746)
Heartgears	20.799 (2.621)	313.421 (62.440)

When compared regarding parsing times, the "Three.js" loader also outperformed the one presented in this work (Table 7). By analyzing the parsing functions of each work, it was discovered that the bottleneck of this work's loader was the need for computing the normals when each face of the geometry was created. In "Three.js" loader, this apparently is already done automatically by the data structures utilized.

Table 7. Time taken by each loader to parse models in PC (average time in ms).

Model	"Three.js" loader PC Average (Standard Deviation)	Our loader PC Average (Standard Deviation)
Tetrahedron	0.119 (0.069)	0.171 (0.137)
Rhombicuboctahedron	0.287 (0.132)	0.423 (0.175)
Dodecahedron chain	24.834 (11.513)	42.404 (17.966)
Heartgears	127.893 (28.460)	308.822 (73.671)

It is important to notice that, during tests, it was discovered that the "Three.js" loader uses a Javascript API called TextDecoder, does not working on all Desktop browsers [9], and by the tests performed, it does not work on mobile devices as well. However, it is a problem that can be fixed, since there are libraries which implement API and give support to such browsers. While there is a loader already in the "Three.js" source code which is more efficient, some opportunities to improve such loader were identified while it was analyzed. It can be further optimized if a different function is used to separate the XML nodes, and support to textures and colors can also be added.

4.4 Printing Simulator Analysis

Since the objective of the final product of the simulation was to run both on mobile and PC, tests were conducted to assert that it could effectively run on those platforms. That is, besides loading and displaying the models, the device where the simulator runs must be able to simulate slicing in a timely fashion, so the user can use it without waiting for results.

In such a way, tests were conducted to determine the speed in which the devices could calculate each slice. Table 8 shows the time it takes for each slice to be computed; each average and standard deviation were taken from 100 samples, except for the Heartgears model which were tested on the mobile devices. For this model, in both phone and tablet, only 10 samples were run due to the long time it took to compute each slice, and the fact that, sometimes, the browser froze while computations were being made.

Table 8. Slicing times (average time in ms).

Model	PC (Standard Deviation)	Phone (Standard Deviation)	Tablet (Standard Deviation)
Tetrahedron	1.248 (0.878)	3.115 (3.337)	7.220 (6.766)
Rhombicuboctahedron	1.836 (0.914)	6.745 (5.894)	11.899 (5.584)
Dodecahedron chain	637.616 (129.334)	1002.773 (234.394)	3360.233 (872.014)
Heartgears	2677.336 (546.256)	8075.494 (2468.433)	19855.443 (2078.619)

Given that the most complex models take too long for the slices to be computed, a strategy was devised to minimize such effects. Each slice is only computed once when the application is used, and a cache is established, so if the user so desires, the simulation can be viewed smoothly after its first execution. Nevertheless, the caching strategy did not solve one problem: complex models cannot be easily simulated through the use of the CSG intersection operation, especially on mobile devices which sometimes froze when the simulation was running. This leaves two options for the simulations to be effective on mobile devices: either optimize the used functions, or precompute the simulation and leave it available as a source file that the final user will use without noticing.

From these two options, the latter is considered to be a better option for incrementing this work on the future, given that it would also make the voxelization of the models possible regardless of the time that would take for the simulation to run, and the results would be available in a fast manner both on PC and mobile.

5 Conclusion

The 3MF model file loader and visualizer were successfully developed and validated through the implemented printing simulator. Meanwhile, the tests conducted were important for properly comparing the STL format and the new 3MF format, not only for their capabilities, but also regarding their possible use in a web application.

When considering file sizes and capabilities, the 3MF is better suited because it has improved scalability, extensibility, and support for textures and colors. For instance, the Heartgears binary STL model is 4.6 times bigger than its 3MF counterpart, and it only contains a triangulated model.

Nonetheless, one can argue that, due to greater loading times, the binary STL file is preferable when considering mobile devices, since in worst case it can take approximately 38.6 times more to load a complex 3MF model than it takes to load the same model in binary STL (as the Heartgears model in the tablet). However, the analysis made when comparing our loader versus the "Three.js" loader showed some points where our loader can be substantially optimized, and as such the loading times for the 3MF format can be greatly reduced, effectively nullifying this argument in the future.

As for the mobile uses for the simulator, it is clear that more optimized approaches will be needed in the future, since the gap between mobile and desktop is still enough to make some applications instances, as in the simulator when working with large models, impractical. The tests showed that a model can take up to almost 20 s for just one slice to be produced in mobile, and in some cases the tab simply crashed without the simulation running.

Regarding the open source contributions of this work, the fact that a more efficient loader already exists at the "Three.js" source code does not mean that there will not be any contribution. Support to texture and color can be added to it, and opportunities for improving it were also identified while its code was being analyzed. Therefore, a contribution to the open source community is still possible.

Acknowledgments. The research results reported in this paper have been partly funded by a R&D project between HP Brazil R&D division and UFPE originated from tax exemption (IPI-Law number 8.248, of 1991 and later updates).

References

1. WebGL: OpenGL ES 2.0 for the Web. Khronos Group, https://www.khronos.org/webgl/. Accessed 16 Oct 2015
2. 3D Printing Has Expanded Faster Than Expected, http://bit.ly/1LmQUI2. Accessed 16 Oct 2015
3. 3MF Specification, http://3mf.io/what-is-3mf/3mf-specification. Accessed 16 Oct 2015
4. Babylon.js, http://www.babylonjs.com/. Accessed 14 Jan 2016
5. Constructive Solid Geometry with CSG.js, http://learningthreejs.com/blog/2011/12/10/constructive-solid-geometry-with-csg-js/. Accessed 16 Oct 2015
6. CSG.js. https://github.com/evanw/csg.js/. Accessed 16 Oct 2015
7. JSzip, https://github.com/Stuk/jszip. Accessed 1 July 2016

8. OpenJSCad, http://openjscad.org/. Accessed 16 Oct 2016
9. TextDecoder, https://developer.mozilla.org/en-US/docs/Web/API/TextDecoder/decode. Accessed 14 Jan 2016
10. Three.js, A Javascript 3D library, http://threejs.org/. Accessed 16 Oct 2015
11. ThreeMFViewer. https://github.com/jvfl/ThreeMFViewer. Accessed 15 Jan 2016
12. What is 3D printing? http://3dprinting.com/what-is-3d-printing/. Accessed 16 Oct 2015
13. What is 3MF? http://3mf.io/what-is-3mf/. Accessed 16 Oct 2015
14. zip.js A JavaScript library to zip and unzip files, https://gildaslormeau.github.io/zip.js/. Accessed 15 Jan 2016
15. Bandyopadhyay, A., Bose, S., Das, S.: 3D printing of biomaterials. MRS Bull. **40**(02), 108–115 (2015)
16. Chiu, W., Tan, S.: Multiple material objects: from cad representation to data format for rapid prototyping. Comput.-Aid. Des. **32**(12), 707–717 (2000)
17. Dudek, P.: Fdm 3D printing technology in manufacturing composite elements. Arch. Metall. Mater. **58**(4), 1415–1418 (2013)
18. Ecma, T.: Office open xml (2006)
19. Hague, R., Reeves, P.: Rapid prototyping, tooling and manufacturing, vol. 117. iSmithers Rapra Publishing (2000)
20. Hiller, J.D., Lipson, H.: Stl 2.0: a proposal for a universal multi-material additive manufacturing file format. In: Proceedings of the Solid Freeform Fabrication Symposium, pp. 266–278. No. 1, Citeseer (2009)
21. Jurrens, K.K.: Standards for the rapid prototyping industry. Rapid Prototyping J. **5**(4), 169–178 (1999)
22. Kumar, V., Dutta, D.: An assessment of data formats for layered manufacturing. Adv. Eng. Softw. **28**(3), 151–164 (1997)
23. Rego, N., Koes, D.: 3dmol. js: molecular visualization with webgl. Bioinformatics **31**(8), 1322–1324 (2015)
24. Sawicki, B., Chaber, B.: Efficient visualization of 3D models by web browser. Computing **95**(1), 661–673 (2013)
25. Stroud, I., Xirouchakis, P.: Stl and extensions. Adv. Eng. Softw. **31**(2), 83–95 (2000)