

# PNG as Fast Transmission Format for 3D Computer Graphics in the Web

Daniel Dworak and Maria Pietruszka

**Abstract** This paper focuses on manners of filling the gaps in existing standards that are used in tridimensional Web technologies. We proposed the way of encoding huge 3D data sets in lossless PNG format and use of programmable rendering pipeline to decoding PNG file. It allows to reduce significantly the file with 3D data, time of transmission via Web and time needed to decode the file.

**Keywords** 3D geometry in the Web · 3D data transmission format · WebGL rendering pipeline · GPGPU

## 1 Introduction

Nowadays, every personal computer, notebook or even mobile device has its own graphics processing unit (GPU) and installed newest Web browser, which has an access to GPU via shaders language. It allows to perform complex calculations using GPU to render tridimensional graphics data in real time. GPU shaders can be also used for non-graphics data-parallel computing, for example image, audio and video processing [3], which is called GPGPU—General-Purpose Computing on Graphics Processing Units.

The main problem for 3D graphics in the Web are huge 3D data sets, which contain many models of buildings, trees, etc. Transmission of such data via network requires wideband Internet connection and transmission formats. There are transmission formats for Audio (MP3), Video (H.264), and images (JPEG, PNG) but no

---

D. Dworak (✉) · M. Pietruszka (✉)  
Institute of Information Technology, Lodz University of Technology, Lodz, Poland  
e-mail: 150859@edu.p.lodz.pl  
URL: <http://www.p.lodz.pl>

M. Pietruszka  
e-mail: [maria.pietruszka@p.lodz.pl](mailto:maria.pietruszka@p.lodz.pl)

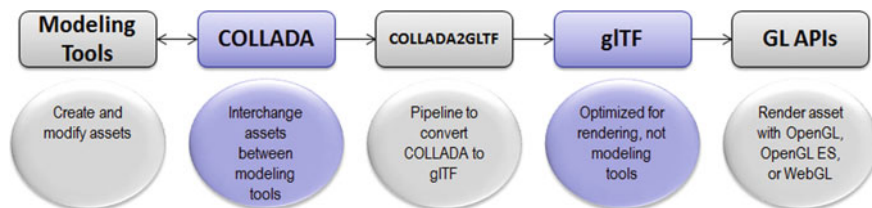
D. Dworak  
Center for Media and Interactivity, Justus Liebig University Giessen, Giessen, Germany

transmission format exists for 3D computer graphics yet. Declarative-3D formats, such as X3D and Collada, specify 3D scene data in a well-structured, human-readable format. However, a text-based meshes data description gives huge files, even for small models that consists of a few thousand polygons. WRL was declarative-3D format of VRML (Virtual Reality Markup Language), first 3D graphics technology for the Web. The X3D technology, as a successor to VRML, accepts description of 3D scene in old WRL format and in new XML format, but in both cases, the additional plug-ins are necessary. Collada is also declarative-3D asset interchange format (DAE), created by the Khronos Group.

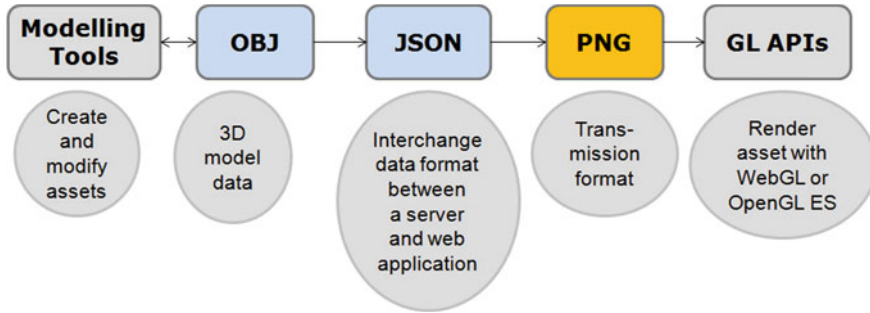
Currently, there are several solutions to convert those formats to modern Web standards without plug-ins. X3DOM is an open-source and runtime framework for 3D graphics for the Web, which allows to include X3D elements as a part of any HTML5 DOM tree [1]. XML3D offers another approach for interactive 3D graphics (encoded in XML), that is not based on any prior standard [11]. More general solution is the Khronos Group's glTF format (WebGL Transmission Format), through which it is possible to use Collada format prepared for current WebGL technology (Fig. 1) [7, 12]. A glTF asset is represented by JSON file (for node hierarchy, materials, cameras, as well as descriptor information for meshes, shaders, animations and other construct), binary files (for geometry and animations, and other buffer-based data), JPEG and PNG image files for textures, and GLSL text file for shader source code.

glTF like X3DOM uses text encoding for structured declarative data (such as transformation groups, materials, cameras, and descriptor information about meshes) and binary encoding for unstructured data for geometry (the vertex attributes of the mesh). Unstructured data usually constitutes more than 95 percent of a file, what affects the data rate. Limper et al. developed two binary 3D mesh formats for declarative 3D approaches: sequential image geometry (SIG) using PNG images as a regular mesh container, and progressive binary geometry (PBG) using a direct binary encoding of mesh data [8]. Our proposition was PNG as transmission format for irregular meshes (Fig. 2) [2]. The research has shown that the 3D data in PNG files reduces its size and time needed to transfer them via Internet.

This paper focuses on the 3D data encoding in PNG file and on data decoding simultaneously. For this purpose, it has been proposed to use a GPU rendering pipeline supported by WebGL technology [4, 9, 10]. Concerning a decoding of



**Fig. 1** Khronos Groups glTF as transmission format for WebGL applications [5]



**Fig. 2** PNG as transmission format for 3D Web applications

large two-dimensional sets, the most important thing is the accessible, high clocked, large memory of GPUs (even up to 8GB for single GPU), which can be used by Web browser easily and fast. The architecture of graphics card allows to perform a lot of operations on the whole data set in one unit of time, which is called calculations parallelism, however it is hard to control the order of tasks, as well. Moreover, there are no dependencies between data elements, therefore it is impossible to addict one value to another, what also makes it hard (or even sometimes not possible) to redesign traditional algorithms.

## 2 Encoding 3D Data to PNG File

Basically, there are five types of data to define 3D model: vertices (space coordinates  $x$ ,  $y$  and  $z$ ), faces (set of vertices indices which creates single polygon), normals (vectors of surfaces orientation for lighting calculations), textures coordinates (called UV's), material properties (like color coefficients and images for texture). We have been decided to use JSON (JavaScript Object Notation) format for storing 3D data. This format is an open standard with user-friendly notation (e.g. readable text) and is used to transmit data from server to client easily. The JSON file has a structure like follows [6]:

**Listing 1.** 3D data for graphics scene:

1. **List of materials:** `"materials" : [{ "DbgIndex" : 0, "DbgName" : "sample", "colorAmbient" : [0.925, 0.807, 0.164], "colorDiffuse" : [0.925, 0.807, 0.164], "colorSpecular" : [0.0, 0.0, 0.0], "transparency" : 0.0, "mapAmbient" : "1.png", "mapDiffuse" : "1.png", }]`
2. **Vertices:** `"vertices": [-5.0, 0.0, 5.0, 5.0, 0.0, 5.0, -5.0, 0.0, -5.0, 5.0, 0.0, -5.0, -5.0, 5.0, ..., -5.0],`
3. **Normals coordinates:** `"normals": [0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, -1.0, ..., 0.0],`

4. **Texture's coordinates:** "uvs": [0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, ..., 1.0],
5. **Faces:** "faces": [42, 0, 2, 3, 0, 9, 11, 10, 0, 0, 0, 42, 3, 1, 0, 0, ..., 11, 11, 11]

The conception of conversion JSON to PNG files is based on saving any 3D data to RGB channels of 2D image (Fig. 3). Then, every value (vertex, normal, UVs) is splitted into integer and fractional parts and stored in R, G, B channels. The Alpha channel ( $A$ ) identifies kind of data: vertex ( $A = 128$ ), faces ( $A = 100$ ), normal ( $A = 50$ ), texture coordinate ( $A = 25$ ), material ( $A = 255$ ).

To store a single vertex coordinate it is necessary to normalize every vertex value to interval from 0 to 255, then we perform calculations as follows:

$$R = \lfloor x_{position} \rfloor \quad (1)$$

$$t = \frac{(x_{position} - R) * 10000}{256}, G = \lfloor t \rfloor \quad (2)$$

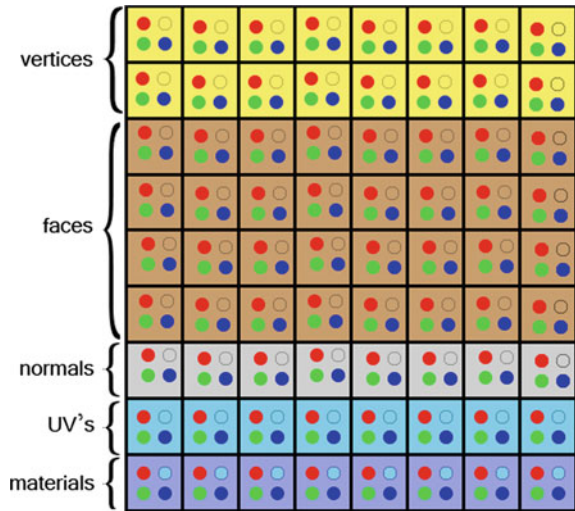
$$B = t - G \quad (3)$$

$$A = 128. \quad (4)$$

where  $\lfloor a \rfloor$  is a floor of  $a$ .

Information about faces are slightly unusually encoded. The face flag and indices are encoded one by one for vertices, material, texture UV's, normals and color. The flag is encoded as follows [6]:

**Fig. 3** Structure of PNG file with geometry—vertices (yellow background), faces (light brown), normals (gray), UVs (light blue) and materials (light violet)



1. bit—face type (0 if quad, 1 if triangle)
2. bit—1 if face has material, 0 otherwise
3. bit—1 if face has UV's, 0 otherwise
4. bit—1 if vertices have UV's, 0 otherwise
5. bit—1 if face has normals, 0 otherwise
6. bit—1 if vertices have normals, 0 otherwise
7. bit—1 if face has colors, 0 otherwise
8. bit—1 if vertices have colors, 0 otherwise

Every face value (flag and indices of vertices, materials, normals and UV's) is an integer number and requires only two channels—R and G:

$$R = \left\lfloor \frac{face_{value}}{256} \right\rfloor \quad (5)$$

$$G = \frac{face_{value}}{256} - R, B = 0 \quad (6)$$

$$A = 100 \quad (7)$$

where  $face_{value}$  is every integer number for faces set.

Storing of normals is similar to faces, but we also store a sign of the value in blue channel.

$$t = \frac{normal_{value} * 10000}{256}, R = \lfloor t \rfloor \quad (8)$$

$$G = t - R \quad (9)$$

$$A = 50 \quad (10)$$

If  $normal_{value} \geq 0$ , then  $B = 255$ , otherwise  $B = 0$ .

And finally, to store a UV's texture value is similar to a normal value, but UV's are not signed.

$$t = \frac{uv_{value} * 10000}{256}, R = \lfloor t \rfloor \quad (11)$$

$$G = t - R, B = 0 \quad (12)$$

$$A = 25 \quad (13)$$

At the end of PNG file, informations about materials are saved. There has been proposed JSON based sequence of properties like: materials' name, colors, textures' names or transparency. A textual information (for example the name of a texture map) is stored in ASCII code—one character in single pixel's channel but material's colors—as RGB pixels.

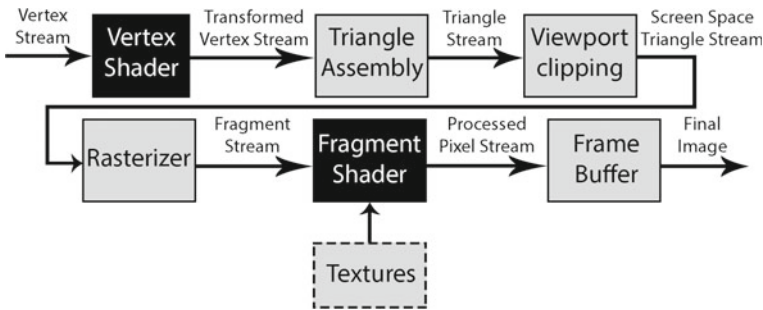
### 3 Decoding 2D Data Sets to 3D Geometry

The 3D data saved in PNG file are decoded according to equations as follows:

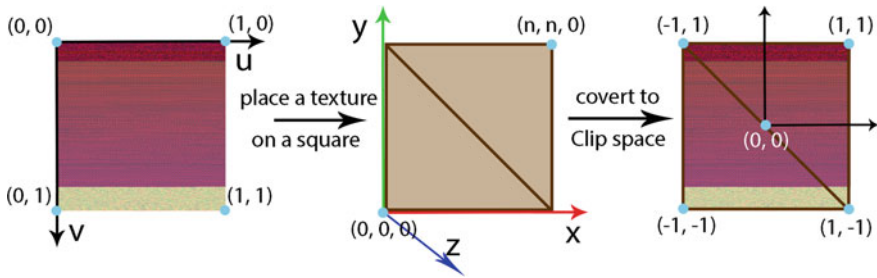
$$\begin{aligned} Vertex_{value} &= R + \frac{G * 256 + B}{10000}, & Face_{value} &= R + G * 256, \\ Normal_{value} &= \frac{(R + G * 256)}{10000}, & UV_{value} &= \frac{R + G * 256}{10000}. \end{aligned} \quad (14)$$

Next, they need to be saved in Vertex Buffer, from where they are transferred to rendering pipeline. The current paradigm for the Web is programmable rendering pipeline (Fig. 4). However, there is no ability to use the latest technologies which are offered by graphics cards (using by 20 % of users), because the Web application has to be run on the great number of computers. Nowadays it is possible to program two shaders. Vertex Shader processes a single vertex, while Fragment Shader a single fragment, which corresponds with a pixel of the output file. Data for Vertex Shader comes from Vertex Buffer, to which they are saved by 3D application. The output data of Fragment Shader's computations are sent to Framebuffer—the place from where are taken to display. In WebGL technology exists Frame Buffer Object (FBO), which allows to create a user-defined invisible Framebuffer. It allows to save floating point numbers (processed by Fragment Shader) in such FBO, which contain decoded 3D data (Eq. 14).

In this article, the main goal of operations performed in rendering pipeline is not a displaying the output PNG picture, but decoding data about vertices, faces, normals and UVs saved there (Fig. 5). It can be achieved by mapping PNG picture on a square. In Fragment Shader, the calculations according to equations (14) are performed and the results are saved in textures texels, which are the floating point numbers. The final output of rendering process is directed to invisible Framebuffer, instead of displaying it on the screen.



**Fig. 4** Programmable rendering pipeline for the web



**Fig. 5** Programmable rendering pipeline for decode 3D data from PNG file;  $n$ —dimension of PNG image

This algorithm defines stages of decoding 3D graphic's data stored in PNG file:

- (1) Load PNG file.
- (2) Define parameters of a texture created from a picture saved in PNG file.
- (3) Save data needed to render a square in Vertex Buffer.
- (4) Draw the square, on which an image from PNG file is mapped.
- (5) Save the results from invisible Framebuffer into arrays of geometry's data.

Only the fourth stage is run on GPU, the other stages are run on CPU.

We have implemented this algorithm using WebGL, that brings hardware-accelerated cross platform and hardware independent 3D graphics to the browser without installing additional software (e.g. plugin). For this reasons and compatibility with HTML5 language it is the most popular nowadays. To perform some actions with WebGL, it requires to create the canvas element which is 2D drawing context API.

- Re 1.** After loading the image to CPU, special buffer named Frame Buffer Object (FBO) and a context (on the canvas element) need to be initialized.
- Re 2.** The parameters for wrapping the texture are set to *GL\_CLAMP\_TO\_EDGE* and *GL\_NEAREST* for mapping; the rest are defaults.
- Re 3.** The square with dimensions of the PNG image is created from two triangles (Fig. 5). The origin of the image is located in the upper left corner, while the origin of scene's coordinates is located in the bottom left corner, therefore to define the right texture coordinates it is necessary to mirror vertical axis. Then, a vertex located in (0,0,0) corresponds with a UV (0,1), whereas a vertex ( $n,n,0$ ) corresponds with UV (1,0).
- Re 4.** In WebGL, it is possible to use *drawArray* function, which renders triangle primitives from the array to binded context *< canvas >* element. Then, the programmable rendering pipeline is launched, for which it is needed to create Vertex Shader and Fragment Shader (Fig. 5). Moreover, it is possible to build a lookup table to ensure high efficiency while determining position of each pixel.

In WebGL technology, the Shaders are programmed in the OpenGL Shading Language (GLSL) [9]. The Vertex Shader, in case of our researches, computes

textures coordinates (Listing 2). The input data are position and UVs coordinates for the current vertex and dimension of PNG image (*u\_resolution*). The space position coordinates (eg. x, y and z) needs to be transposed to Clip space coordinates (interval [1; 1]) due to requirements of *gl\_Position* variable. At the end, the *gl\_Position* and *v\_texCoord* are redirected to Fragment Shader. Input data for Fragment Shader is also the texture image (*u\_image*). Next, according to alpha value (converted to interval [0.0, 1.0]) and Eqs. (14) a data about 3D scene are calculated using Fragment Shader (Listing 3).

**Listing 2.** An example of Vertex Shader that handles the texture and passes it to Fragment Shader

```
<script id="vertexShader" type="x-shader/x-vertex">
attribute vec2 a_position; //position of the current vertex/pointer
attribute vec2 a_texCoord; //texture's UV's
uniform vec2 u_resolution; //texture's resolution
varying vec2 v_texCoord; //coordinates for Fragment Shader

void main() {
    vec2 zeroToOne = a_position / u_resolution; //convert pixels'
                                                rectangle to [0.0, 1.0]
    vec2 zeroToTwo = zeroToOne * 2.0; // convert from 0->1 to 0->2
    vec2 clipSpace = zeroToTwo - 1.0; //convert from 0->2 to -1->+1 (clipSpace)
    gl_Position = vec4(clipSpace, 0, 1); //convert the gl_Position to Clip space
    v_texCoord = vec2( a_texCoord.s, a_texCoord.t); //texCoord for Fragment Shader
} </script>
```

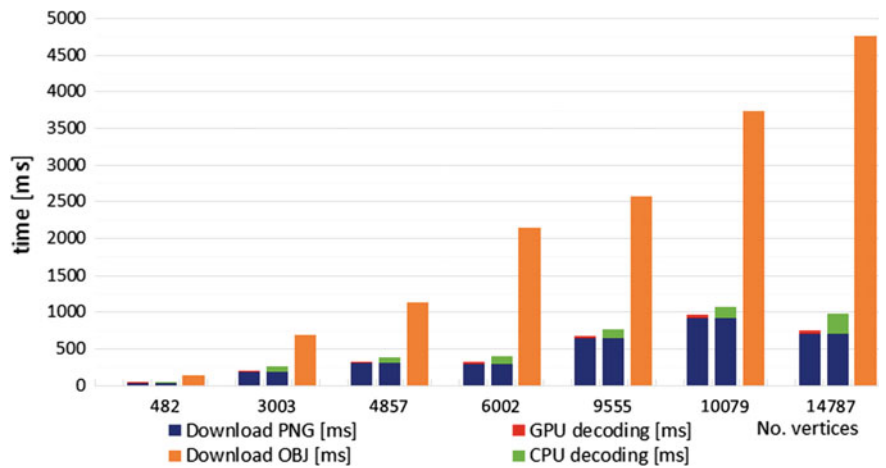
**Listing 3.** An example of Fragment Shader—changing equations due to alpha value for vertices, faces, normals and texture's coordinates.

```
<script id="fragmentShader2" type="x-shader/x-vertex">
uniform sampler2D u_image; //sampler2D allows to sample pixel colours
                             from a 2D texture
varying vec2 v_texCoord; // the texCoords from the Vertex Shader.

void main() {
    vec4 finalColor = texture2D(u_image, v_texCoord);
    // finalColor[i] - array of BGRA values, where i=0 is B channel,
    // i = 1 is G, i = 2 is R, i = 3 is A /
    if (finalColor[3] == 0.50) { //Vertices, alpha = 128
        float float_n = float(finalColor[2] + (finalColor[1]*256.0 +
                                                finalColor[0])/10000.0); }
    else if (finalColor[3] == 0.40) { //Faces, alpha = 100
        float float_n = float(finalColor[2] + finalColor[1]*256.0); }
    else if (finalColor[3] == 0.20) { //Normals, alpha = 50
        float float_n = float((finalColor[0] + finalColor[1]*256.0)/10000.0); }
    else if (finalColor[3] == 0.10) { //UV's, alpha = 25
        float float_n = float((finalColor[0] + finalColor[1]*256.0)/10000.0); }
    gl_FragColor = encode_float(float_n * 255.0); //output for CPU }
</script>
```

**Re 5.** Outputs of rendering are saved in invisible canvas, read by *readPixels* WebGL function and forwarded to arrays, which are used to create vertex buffers for 3D scene.

Present implementation of WebGL standard does not support Geometry Shader, which could allow to change geometry of particular 3D data. The texture returned by the Fragment Shader contains of float values which are saved in invisible canvas, read by *readPixels* WebGL function and forwarded to arrays of floats, which are used to create vertex buffers for 3D scene.



**Fig. 6** Downloading (OBJ and PNG) and decoding (PNG) time comparison on CPU and GPU for different complexity of geometry

4 CPU and GPU Benchmarks

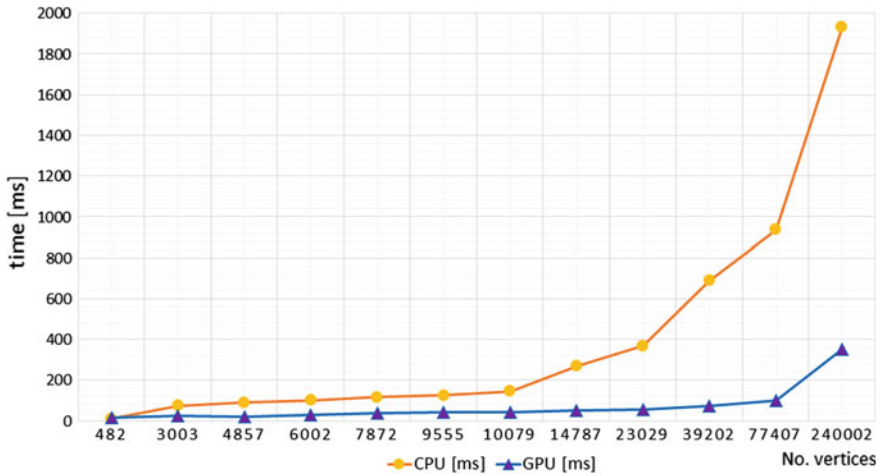
The authors’ idea of saving geometry in PNG files is likely to decrease the size of files. On the other hand, there are additional calculations that need to be performed on client side during decoding process. Of course, it takes much less time to transfer smaller file via Web (Fig. 6.). For example, a model with 77 407 vertices saved in OBJ file (13 MB) downloads about 25 s, while model encoded by authors’ algorithm in PNG (2.25 MB) only 4.5 s (Table 1). For the same PNG file, a period of time that is needed to decode a data set is also short (less than 1 s), but GPU decoding is nine times faster than CPU. As Table 1, Figs. 6, and 7 show, there is a big change between calculations performed on CPU and GPU.<sup>1</sup>

Concerning small files, CPU processing is a bit faster than GPU. It results in a constant time that is needed to create context, trigger shaders and allocate a memory on a GPU. In fact, there are bigger changes for more complex geometry. In case of CPU processing, time grows quadratically—for two times larger input files required time doubles. On the contrary, GPGPU’s dependency between file’s size and time is logarithmic like, what is caused by parallelism of tasks on huge data sets. There is no such growth of appropriate time like during traditional computing. What is more, a processing time is even the same but for ten times larger input picture.

<sup>1</sup>Technical specification: CPU Intel i7-4702HQ 2.2GHz, GPU Nvidia K110M 2GB.

**Table 1** Downloading (4Mb/s) and decoding time comparison on CPU and GPU for different complexity of geometry saved in PNG and OBJ file

No. vertices	File's size (KB)		Downloading (ms)		Decoding (ms)	
	OBJ	PNG	OBJ	PNG	CPU	GPU
482	69	17	135	33	10	14
3003	355	93	693	182	75	23
4857	578	155	1129	303	89	22
6002	1102	150	2152	293	101	28
7872	1126	318	2199	621	117	37
9555	1321	328	2580	641	125	41
10079	1916	471	3742	920	146	43
14787	2442	363	4770	709	270	49
23029	3428	838	6695	1637	368	55
39202	6882	1377	13441	2689	688	74
<b>77407</b>	<b>12971</b>	<b>2308</b>	<b>25334</b>	<b>4508</b>	<b>936</b>	<b>102</b>
240002	21852	3595	42680	7021	1932	351
1120000	50812	7884	99242	15398	6891	533

**Fig. 7** Decoding time comparison on CPU and GPU for different complexity of geometry

## 5 Conclusions

The virtual 3D reconstructions using portable and crossplatform technologies are burdened with many restrictions. Now, most of modern 3D technologies are not standards and there are many aspects of improving them. Despite our proposed solutions are in development stage, they are promising so far. This fact can be confirmed by

achieved results—the reduction of file’s size and time of decoding is over few times. There are many reasons to continue researches in this way, but Internet technologies (like WebGL) are demanding, needing many techniques of optimization to reach the main aim.

The analysis of reached results (Table 1, Figs. 6 and 7.) confirms that proposed solution highly reduces required time to decode data sets, what also makes it faster to create a geometry and display it on the screen. It is easy to notice that CPU processing has almost quadratic complexity, so obligatory time is strictly connected with the size of input data. On the other hand, there is no such dependency while GPU processing is performed—even when the input file is several times bigger, there is a change by only few percent. It is also worth to mention that processing time on CPU is shorter in case of really small sets (about few hundred of vertices), what is caused by constant time required to allocate memory and compile shaders while using GPU. Summing up, the application of GPGPU allowed to reduce appropriate time needed to decode a data encoded in PNG file even few times. What is more, it makes sense to carry on this kind of computing for relatively huge data sets, what is not a problem because of trends in modern tridimensional technologies.

**Acknowledgments** The research is partially performed within the project “Virtual reconstructions in transitional research environments—the Web portal: Palaces and Parks in former East Prussia” supported by Liebnitz Gemeinschaft in the years 2013–2016.

## References

1. Behr, J., et al.: X3DOM: a DOM-based HTML5/X3D integration model. In: Proceedings of the 14th International Conference on Web3D Technology (Web3D 09), ACM, pp. 127–135 (2009)
2. Dworak, D., Pietruszka, M.: Fast Encoding of Huge 3D Data Sets in Lossless PNG Format. *Advances in Intelligent Systems and Computing: New Research in Multimedia and Internet Systems*, vol. 314, 15–24. Springer (2015)
3. Fung J., Mann S.: Computer Vision Signal Processing on Graphics Processing Units. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004), Montreal, Quebec, Canada, pp. 93–96 (May 2004)
4. Goddeke D.: GPGPU Basic Math Tutorial (online: 15.04.2016). <http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html>. Accessed 18 April 2016
5. <https://github.com/KhronosGroup/glTF/wiki/converter>. Accessed 18 April 2016
6. [https://github.com/mrdoob/three.js/blob/master/utils/converter/obj/convert\\_obj\\_three.py](https://github.com/mrdoob/three.js/blob/master/utils/converter/obj/convert_obj_three.py). Accessed 18 April 2016
7. Khronos: Efficient, Interoperable Transmission of 3D Scenes and Models. <https://www.khronos.org/glTF>. Accessed 18 April 2016
8. Limper, M., Jung, Y., Sturm, T., Franke, T., Schwenk, K., Kuijper, A.: Fast, progressive loading of binary-encoded declarative -3D web content. In: *Computer Graphics and Applications*, IEEE, vol. 33, pp. 26–36 (2013)
9. Rost, J., Licea-Kane, B.: *OpenGL Shading Language*. 3rd edn. Addison-Wesley (2010)
10. Shreiner, D., Angel, E.: *Interactive Computer Graphics with WebGL: Global Edition*. Pearson Education (2014)
11. Sons K., et al.: XML3D: Interactive 3D graphics for the web. In: Proceedings of the 15th International Conference on Web3D Technology (Web3D 10), ACM, pp. 175–184 (2010)

12. Trevett, N.: 3D Transmission Format, NVIDIA (June 2013)

Multimedia and Network Information Systems  
Proceedings of the 10th International Conference MISSI  
2016

Zgrzywa, A.; Choroś, K.; Sieminski, A. (Eds.)  
2017, XIV, 432 p. 137 illus., 91 illus. in color., Softcover  
ISBN: 978-3-319-43981-5