Visualizing Large Image Datasets in 3D Using WebGL and Media Fragments

Charles-Frederik Hollemeersch*
Ghent University - IBBT
ELIS - Multimedia Lab
Ghent, Belgium

Bart Pieters
Ghent University - IBBT
ELIS - Multimedia Lab
Ghent, Belgium

Aljosha Demeulemeester
Ghent University - IBBT
ELIS - Multimedia Lab
Ghent, Belgium

Davy Van Deursen
Ghent University - IBBT
ELIS - Multimedia Lab
Ghent, Belgium

Peter Lambert
Ghent University - IBBT
ELIS - Multimedia Lab
Ghent, Belgium

Rik Van de Walle
Ghent University - IBBT
ELIS - Multimedia Lab
Ghent, Belgium

ABSTRACT

The recent standardization of WebGL opened new possibilities for graphically-intensive web-based applications. In this paper, we show how we can interactively visualize very large texture datasets (in the order of gigapixels) on arbitrary 3D geometry using WebGL and JavaScript. Our results show that real-time performance can be achieved on current-generation hardware and browsers.

Categories and Subject Descriptors

I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism - Color, shading, shadowing and texture

Keywords

WebGL, Streaming, Visualization

1. INTRODUCTION

The new HTML5 [3] standard adds many valuable features for web developers. Web applications using HTML5 are visually richer, more responsive and offer greater platform independence. Instead of relying on vendor and platform specific plugins, standardized technologies such as 2D canvas and 3D WebGL [5] allow graphics operations to be scripted directly from within JavaScript. WebGL is a JavaScript binding of the existing OpenGL-ES standard which offers developers a lightweight (OpenGL-ES was originally meant for embedded applications) but flexible 3D graphics API. Both Chrome and Firefox support WebGL in their public releases while both Opera and Safari are working on WebGL support. WebGL is also being considered for mobile platforms. For example, the beta version of Firefox for Android has experimental WebGL support.

Besides WebGL, the media fragments specification is another new standard which supports rich multimedia applications [7]. This specification describes a standardized URI scheme for accessing fragments of a media resource. For example, individual audio or video fragments can be addressed in a longer media sequence. Not limited to temporal fragments, the schema also allows accessing spatial fragments within an image or video frame as well as addressing several tracks within a single resource.

Visualizing high-resolution image data sets is a common problem in many applications. GIS, biology, archeology, heritage, and educational applications all have benefited from efficiently acquiring and accessing large image data sets [1]. When extending such applications to 3D, new problems arise such as determining which data is needed at what resolution. Existing online 3D solutions such as Google EarthView[1] or Nokia Maps 3D[2] again rely on browser plugins to generate the 3D visuals. In this paper we will describe our system that allows visualizing large image datasets (referred to as textures in a 3D visualization context) solely using the WebGl standard.

2. OVERVIEW OF THE SYSTEM

Figure 1 shows a screenshot of our application. A public demo is accessible at `http://multimedialab.elis.ugent.be/webgl_demo/`. This demo visualizes a 15 gigapixel orthophoto mapped onto a 3D mesh that was generated from height data. Note that our system is not limited to planar or landscape-like geometry but supports arbitrary 3D models such as buildings and indoor scenes. To avoid having to download the whole dataset (75 gigabytes uncompressed and two gigabytes compressed) when the user initially visits our site, we dynamically stream only the visible portions of the dataset. This allows our system to generate images as shown in Figure 1 with only a few seconds of loading time. After initial loading, the user can then freely explore the scene while additional texture data is dynamically downloaded using asynchronous image requests only when it becomes visible.

Figure 2 shows the architecture of our application. The server side is split in two parts, one part hosts static resources (i.e. html files, JavaScript files, ...), the other part

*Contact e-mail: charlesfrederik.hollemeersch@ugent.be

Copyright is held by the author/owner(s).

WWW2012 Developer Track, April 18-20, 2012, Lyon, France.

[1]`http://maps.google.com/earthview/`
[2]`http://maps.nokia.com/3D/`

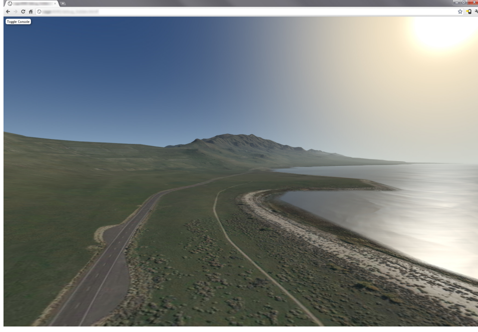

Figure 1: A screenshot of our web site running in Google Chrome.

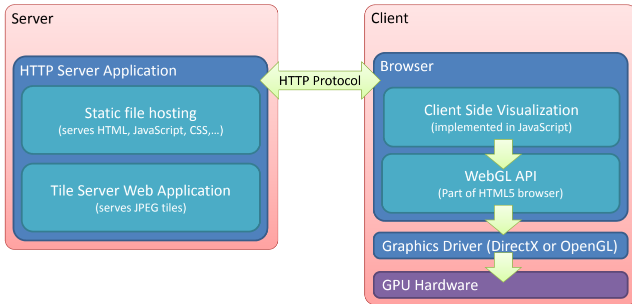

Figure 2: The logical architecture of our system.

then serves the texture tile requests using media fragment query URIs. This part will be discussed in more detail in Section 4. At the browser side, our application runs entirely in JavaScript using WebGL to efficiently offload any computationally intensive tasks to native code or even the GPU.

3. WEBGL IMPLEMENTATION

Our system works by storing the tiles which were downloaded from the server in a client side cache. This cache is physically stored as a WebGL texture object. The tiles present in the cache are managed by our application in JavaScript code. To efficiently render from this cache using WebGL, an additional lookup table, the *tile table*, is needed. The tile table is also stored as a WebGL texture object. The tile table needs to be updated whenever the contents of the cache change. To optimize this update, the texture is filled using WebGL render-to-texture commands instead of manually modifying the pixel values from JavaScript. Figure 3 shows the process of rendering a single view in our system using the cache and tile table. We refer to our publication on the desktop version of our system [4] for a detailed technical explanation of the rendering process using these two data structures.

One of the biggest challenges when switching from 2D image zooming applications to arbitrary 3D meshes is determining the set of tiles that need to be present in the cache. Due to the presence of arbitrary texture coordinates there is no strict analytical relationship between the visualized pixels and the corresponding resolution needed of the gigapixel image. Additionally, further away portions of the dataset may be occluded by geometry closer to the camera. E.g. in Figure 1, portions of the dataset hidden behind the hill do not need to be loaded by the system. To overcome these issues, our system works by rendering an additional low resolution view of the scene. Instead of outputting shaded colors in this low resolution view, we output the addresses of the tiles that need to be present to visualise this view. The rendered pixels are then read back into a JavaScript array and analyzed by looping over them. Any addresses encountered which are not present in the cache will then be requested from the server through an asynchronous image request. As we will show in Section 5, this buffer is sufficiently small that analyzing it using JavaScript does not impose a significant performance impact.

It is important to note that our system, except for analyzing the low-resolution rendered view, does not need to do any data intensive pixel processing operations. Textures arriving from the server are encapsulated in a image object that can be passed to OpenGL without requiring any further processing. As we noted above, our tile table is updated using WebGL drawing commands. As we will show in Section 5, this careful implementation allows us to easily achieve realtime speeds. The JavaScript code is in fact nothing more than a director for the work happening in other parts of the browser and graphics driver.

4. USE OF MEDIA FRAGMENT URIS

As discussed in the previous section, our application dynamically requests multi-resolution image tiles from the server. To transfer image tiles, several other technologies and URI formats have been proposed in the past. Examples are the Internet Imaging Protocol [2] and the Deepzoom [6] URI formats. However, we opted for the media fragments URI format. First, using a W3C standard should ensure that in the future our browser based application can easily work independently of any particular image server. Second, the media fragment URI format uses pixel based requests instead of requesting data using tile numbers. This is important since it helps to further hide application specific details such as the tile sizes and layouts from the server. For example, to ensure fast and high-quality texture filtering, our system can use overlapping tiles (we refer to [4] for more details on filtering using overlapped tiles). By using per-pixel addressing, this visualization-specific detail can be hidden from the server. To map our multi-resolution axis to media fragments we used the track parameter. For example, the following URI `http://www.example.com/example.jpages?xywh=476,596,128,128&track=1` requests the 120×120 tile with index $(4, 5)$ on resolution level 1 with a 4 pixel border overlap.

5. RESULTS

We now briefly discuss the performance of our system on a variety of browsers. We tested our system on the latest (release) versions of Chrome(v16) and Firefox (v10). For opera we used Opera Next (v12-alpha). Note that on Opera our system does not generate valid output. This seems to be related with initializing textures from JavaScript typed arrays as opposed to initializing them from image dom objects. Although this results in invalid output, it does not affect some subsystems of our application and thus we included the results in our comparison.

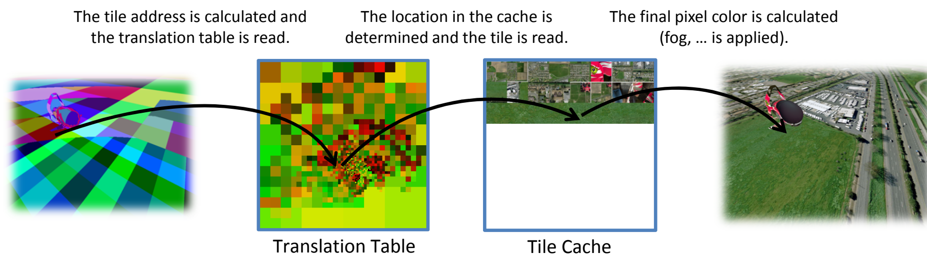

Figure 3: The steps needed to render a picture using the tile table and tile cache.

Table 1: Execution times (in milliseconds) of the different subsystems. Results were measured on an Intel Core2 2.4GHz processor with an NVIDIA GeForce GTX 480 GPU.

System	Chrome	Firefox	Opera
Render off-screen buffer	0.03	0.17	0.07
Read back buffer	10	2.9	6.5
Analyze buffer	1.4	2	1.2
Render main view	0.04	0.13	0.21
Total per frame	11.47	5.2	7.98
Single tile request	168	422	260
Single tile load	2	1.3	1.5
Total per tile	170	423.3	261.5

Table 1 shows the execution times of the different steps needed to render a single frame. The off-screen buffer render phase is the time it takes to render the low resolution view that will be analyzed by JavaScript. The read back phase is the time it takes to read back the data from the GPU so it can be accessed in JavaScript. The analyze phase is the time it takes to effectively loop over the pixels and generate http requests for any missing tiles. Finally the main visuals phase is the time it takes to render the final output that is presented to the user (i.e. the texture model and background sky and fog effects). The last two rows then show the tile request time (time elapsed between requesting a new tile and our JavaScript code asynchronously receiving the tile through a callback) and the tile load time (time elapsed to upload the tile to the GPU cache).

From these performance results, we can draw some interesting conclusions on the design of the different browsers. We observe that Chrome still excels in JavaScript intensive tasks such as setting up the draw commands. However, note that the other two browsers perform a lot better on the read back phase. This is probably caused by Chrome's out-of-process WebGL architecture, which makes the latency of synchronizing with the external process which in turns has to synchronize with the GPU a rather expensive operation.

Besides looking at pure performance results, it is also interesting to look at the overall responsiveness when navigating the scene. Here Chrome is the clear winner over Firefox (we do not consider Opera here due to its invalid output). The main reason for this is that while Chrome takes 168 milliseconds to request a page, the actual request does not block the main thread. I.e. the user can still move around the camera and interact with the page during this period. In comparison, Firefox scores much worse here. First it takes around 422 milliseconds to request a page and hand it over to our JavaScript code. Second, and more annoying, it blocks the main user interface for at least some portion of this time. Considering that we request several pages per second, this results in a very sluggish, almost unusable, performance when many pages have to be delivered to the cache.

6. CONCLUSIONS AND DISCUSSION

In this paper we have shown how modern web technologies can be used to create rich, interactive multimedia applications. We have also shown how real-time results can be achieved with current-generation browsers and computers. As developers with background in native applications, the authors are genuinely surprised about the performance and capabilities of the HTML5 platform. The authors certainly believe that HTML5 combined with WebGL could replace many platform specific native applications with web-based counterparts.

7. ACKNOWLEDGMENTS

The research activities that have been described in this paper were funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (IBBT), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWOFlanders), and the European Union.

8. REFERENCES

[1] K. A. Frenkel. Panning for science. *Science*, 330:748–749, November 2010.

[2] Hewlett Packard Company, Live Picture Inc., and Eastman Kodak Company. *Internet Imaging Protocol v1.0.5*, 1997.

[3] I. Hickson, editor. *HTML5 A vocabulary and associated APIs for HTML and XHTML*.

[4] C. Hollemeersch, B. Pieters, P. Lambert, and R. Van de Walle. Accelerating virtual texturing using cuda. In W. Engel, editor, *Gpu Pro: Advanced Rendering Techniques*, chapter 10.2, pages 623–641. A K Peters., 2010.

[5] C. Marrin, editor. *WebGL Specification*. Khronos Group, 2011.

[6] Microsoft Corporation. *Deep Zoom File Format Overview*, 2012. http://msdn.microsoft.com/en-us/library/cc645077(v=vs.95).aspx.

[7] R. Troncy, E. Mannens, S. Pfeiffer, and D. Van Deursen, editors. *Media Fragments URI 1.0*. W3C, 2011.