Optimizing Data Transfer for WebGL Applications

Thomas Kjeldsen

November 15, 2012

1 Introduction

WebGL is a technology that has pushed the limits of content that can be published on the web. For example, Fig. 1 shows that WebGL allows us to render very complex scenes with hundreds of thousands of polygons in real-time directly in a browser window. One challenge is now that large amounts of geometry data must be transferred over a network connection prior to rendering of such detailed scenes.

With current broadband connections users expect smooth browsing experience where web pages load almost immediately. If the loading time exceeds a few seconds the audience often lose interest and proceed to another website [1]. This timeframe turns out to be hard to reach for some WebGL applications. As shown in Fig. 1, the size of raw vertex data can easily be on the order of tens of megabytes. With a reasonably fast connection with a 10 Mbit/s bandwidth, it takes around one second to load each megabyte of vertex data. On the other hand, JavaScript engines used in modern browsers can process data at a much higher rate, and, hence, it should be possible to obtain faster loading times if we somehow can decrease the amount of data transferred over network at the expense of some post-processing at the client side.

In this paper we will go through various techniques that can be used to optimize the loading time for large amounts of vertex data. Furthermore, we will demonstrate how data can be cached so that it does not need to be reloaded between browser sessions. We assume that the reader has basic knowledge about OpenGL vertexbuffers, JavaScript, and Ajax. We will extensively make use of features that are still W3C working drafts so the code may not run on older browsers and functionality may change in the future. All the code samples are tested with Google Chrome 20 and Mozilla Firefox 16.

2 Basic WebGL

In the rest of the paper, we will use the code structure listed below. When the page loads, window.onload is triggered and sets up the WebGL context, creates

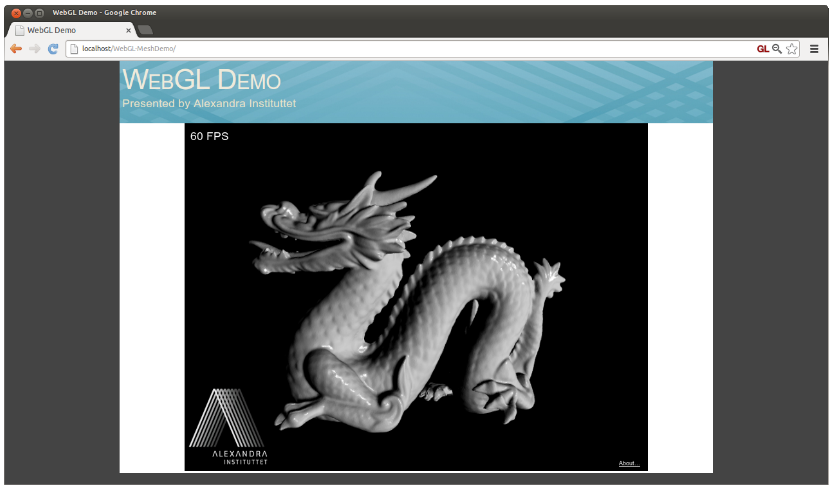

WebGL Demo - Google Chrome
WebGL Demo
localhost/WebGL-MeshDemo/
WEBGL DEMO
Presented by Alexandra Instituttet
60 FPS
ALEXANDRA INSTITUTTET
About...

Figure 1: The Stanford dragon consists of 871414 triangles. Models with this polygon count are straightforwardly rendered in real time in WebGL. A big challenge, however, is how to load the massive amounts of vertex data for such detailed models. For example, the model shown in the figure uses 60 megabytes of vertex position and normal data.

the vertexbuffer, and compiles shaders. Next the vertex data is loaded with an asynchronous XMLHttpRequest. We attach eventhandlers to the request in order to show the load progress. When the request completes we use the response to fill the vertexbuffer, and, finally, we draw the scene.

```
// Globals
var gl;                // GL context
var vertexbuffer;      // GL vertexbuffer object

function openProgress() {
    /* Open a progress dialog */
};

function runProgress(e) {
    /* Update progress dialog */
};

function closeProgress() {
    /* Close progress dialog */
};

function loadData(filename) {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", filename);
    xhr.onload = function(){
        var vertexdata;
        /* use this.response to construct vertexdata */
        gl.bindBuffer(gl.ARRAY_BUFFER, vertexbuffer);
        gl.bufferData(gl.ARRAY_BUFFER, vertexdata, gl.STATIC_DRAW);
        drawScene();
        closeProgress();
    };
    xhr.onprogress = runProgress;
    xhr.onloadstart = openProgress;
    xhr.send();
};

function drawScene() {
    /* Render the scene */
};

window.onload = function(){
    /* Setup GL context, create vertexbuffer, and compile shaders */
    loadData("path/to/vertexdatafile");
};
```

The complete source code is available at our website [2].

The main topic of the present paper is how to write the loadData function in order to load the vertexbuffer most efficiently. If we have stored the vertex data as a raw binary file on our webserver, we can easily construct the vertexbuffer as follows

```
function loadBinaryData(filename)
{
    var xhr = new XMLHttpRequest();
    xhr.open('GET', filename);
    xhr.responseType = "arraybuffer";
    xhr.onload = function(){
        // this.response is now a generic binary buffer which
        // we can interpret as 32 bit floating point numbers.
        var vertexdata = new Float32Array(this.response);
        gl.bindBuffer(gl.ARRAY_BUFFER, vertexbuffer);
        gl.bufferData(gl.ARRAY_BUFFER, vertexdata, gl.STATIC_DRAW);
        drawScene();
    };
    xhr.onprogress = runProgress;
    xhr.onloadstart = openProgress;
    xhr.send();
}
```

Notice that we set the response type to "arraybuffer" to indicate that we expect binary data, and, correspondingly, this.response will be a generic binary buffer. It is not strictly necessary to create the floating point view of the buffer. We could just use the generic buffer in the bufferData call. The actual specification of how the content of the vertexbuffer should be interpreted is set with the vertexAttribPointer method.

One note about method listed above is that we must ensure that client uses the same byte ordering (endianness) as is used for the binary file. One workaround to this problem could be to convert the binary file to a text file with a string representation of one floating point number on each line. We can load such a datafile with the following method.

```
function loadAsciiData(filename)
{
    var xhr = new XMLHttpRequest();
    xhr.open("GET", filename);
    xhr.onload = function(){
        var vertexdata = new Float32Array(this.response.split("\n"));
        gl.bindBuffer(gl.ARRAY_BUFFER, vertexbuffer);
        gl.bufferData(gl.ARRAY_BUFFER, vertexdata, gl.STATIC_DRAW);
        drawScene();
        closeProgress();
    };
    xhr.onprogress = runProgress;
    xhr.onloadstart = openProgress;
    xhr.send();
}
```

This, however, is likely to increase the size of the datafile by a factor of 2-3 depending on the number of digits printed on each line. Another way to solve the byte ordering problem would be to create a DataView of the arraybuffer and use getFloat32(offset, true) to change byte ordering [3].

3 Example: Danish municipalities

We recently published a WebGL demo that allows the user to inspect data about Danish municipalities interactively [4]. The geometric model used in Fig. 2 consists of 74157 triangles, i.e., 222471 vertices. Each vertex has the following attributes: position (three floats), normal used for shading (three floats), and one texture coordinate used associate the vertex to a specific municipality (one float). The size of the vertex buffer is then $222471 \cdot (3+3+1) \cdot 4\,\mathrm{B} = 5.94\,\mathrm{MB}$. As mentioned in the introduction, it can easily take a couple of seconds to load such amounts of data. If we want faster loading time, we need to decrease the network data transfer. Since almost every vertex is shared between three triangles, the first idea would be perhaps be to store only the unique vertices and use an index buffer to draw the triangles. This would roughly decrease the size of the

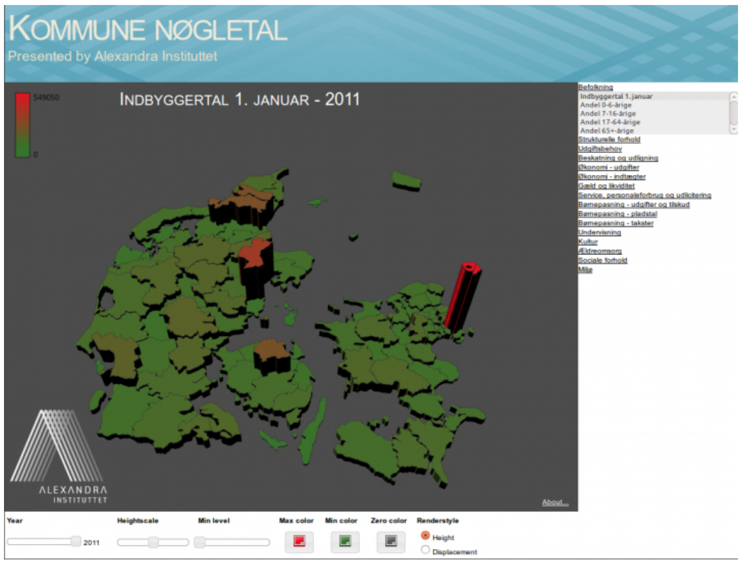


Figure 2: Interactive information visualization of Danish municipalities. The map consists of 74157 polygons.

vertexbuffer to one third with a minor additional cost of the indexbuffer. One problem is that WebGL only supports 16 bit indexbuffers, i.e., it is impossible to index vertexbuffers with more that 65536 vertices. Additionally, an indexbuffer does not take advantage of the fact that many vertices share the same normal and the same texture coordinate. However, with much duplicated data, it should be possible to reduce the data transfer by standard compression methods which will be discussed in the following sections.

4 Serverside zlib compression

A simple way to compress the vertex data is to let the webserver compress the data on the fly prior to the network transfer. It is possible to enable serverside zlib compression on an Apache server using the deflate module [5]. The module must be configured to compress binary data as follows

```
deflate.conf:

<IfModule mod_deflate.c>
    AddOutputFilterByType DEFLATE application/octet-stream
</IfModule>
```

Furthermore, we must ensure that the server interprets the datafile as the mime type application/octet-stream. Usually it is sufficient to set the filename extension to ".bin". All modern browsers have built-in support for zlib decompression.

Using this techniques the amount of data transferred over the network is reduced from 5.9 MB to 758 KB with a negligible overhead in compressing and decompressing. The code listed in the previous section does not need to be modified in any way. The main drawback with this method is that you need administrator access to your webserver or that you can convince your admin that it is a good idea to enable the compression module. Another problem is that the server does not know a priori the number of bytes that it needs to send because the compression happens on the fly in several chunks. This may not seem to be a major issue, however, it is necessary to know the transfer size if we want to implement a reliable progress counter.

We may suggest to gzip the binary data and put the gzip'ed file on the server and load the data with something like

```
xhr.open("GET", filename.gz);
xhr.onload = function(){
  // Port zlib to javascript and implement gunzip
  var vertexdata = new Float32Array(gunzip(this.response));
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexbuffer);
  gl.bufferData(gl.ARRAY_BUFFER, vertexdata, gl.STATIC_DRAW);
  drawScene();
}
```

Unfortunately this would require that we port zlib to JavaScript and run the decompression within JavaScript which would probably be somewhat slower than the browser's native zlib support.

5 PNG compression

In this section we will show how we can use a PNG image to transfer binary data. This method relies on the fact that the PNG format is lossless, inherently applies zlib compression, and that all modern browsers have built-in support for PNG decompression. The basic idea is to create a PNG image from the floating point vertex data by encoding the raw bytes as pixel colors. Examples of how to create the image with graphics libraries such as GD and ImageMagick are provided in the Appendix. The size of the image corresponding to the 5.9 MB vertex data used in the previous example is just 757 KB. We upload the image to our webserver and specify its path to the loadData function. Information about how to create an html image element from an XMLHttpRequest can be found in Ref. [6].

When the image has been loaded, we must convert the pixel colors back to the original vertex data. The main steps in the conversion are outlined below

- Create a new canvas element and resize it to fit the image size.
- Draw the image to the canvas.

- Read back the canvas pixels to an arraybuffer.
- Upload the arraybuffer to graphics card.

A minor note about the readback is that the canvas has an alpha channel even if the image that we draw does not have an alpha channel. Consequently, we must remove every fourth entry of the readback buffer in order to restore the original byte sequence. The complete code for loading the PNG encoded vertex data is listed below.

```
function loadPNGData(filename)
{
  // browser prefixing needed for cross-browser compatibility
  window.URL = window.URL || window.webkitURL;

  var xhr = new XMLHttpRequest();
  xhr.open("GET", filename);
  xhr.responseType = "blob";
  xhr.onload = function(){

    var img = document.createElement("img");

    img.onload = function(){

      // Create a new canvas element and resize it
      var canvas2d = document.createElement("canvas");
      canvas2d.width = img.width;
      canvas2d.height = img.height;

      var ctx2d = canvas2d.getContext("2d");

      // Draw the image to the canvas
      ctx2d.drawImage(img,0,0);

      // Read back the canvas pixels
      var imagedata = ctx2d.getImageData(0, 0, img.width, img.height).data;

      // imagedata is now an Uint8Array of length 4*img.width*img.height
      // which contains the RGBA pixel values read from the canvas.
      // Remove alpha channel from each pixel. Reuse the imagedata array.
      for (var i = 0; i < img.width*img.height; i++)
      {
        imagedata[3*i] = imagedata[4*i];
        imagedata[3*i+1] = imagedata[4*i+1];
        imagedata[3*i+2] = imagedata[4*i+2];
      }
      // The first 3*img.width*img.height elements in imagedata are now
      // exactly equal to the raw bytes of the original vertex data
      var vertexdata = imagedata.subarray(0,3*img.width*img.height);

      gl.bindBuffer(gl.ARRAY_BUFFER, vertexbuffer);
      gl.bufferData(gl.ARRAY_BUFFER, vertexdata, gl.STATIC_DRAW);

      drawScene();
      closeProgress();

      // Explicit destruction is required
      window.URL.revokeObjectURL(img.src);
    };
    img.src = window.URL.createObjectURL(this.response);

  };

  xhr.onprogress = runProgress;
  xhr.onloadstart = openProgress;
  xhr.send();
};
```

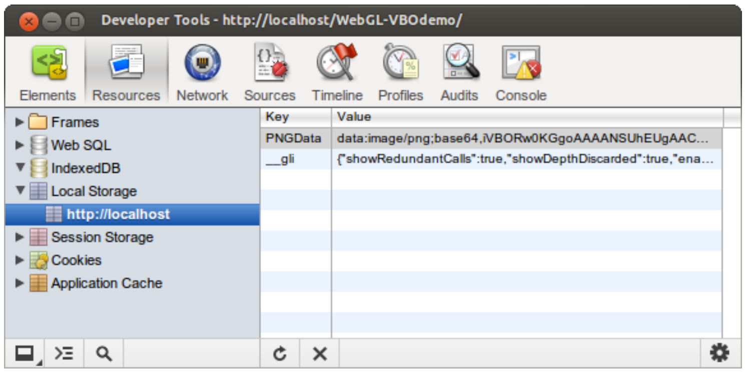
The method listed above may appear to require a lot of post-processing on the client side. However, as stated in the introduction and shown in the benchmark below, the PNG conversion actually turns out to be very fast compared to the time that we save on network transfer.

6 Using web storage

In the previous section, we showed how to encode data in a PNG image to reduce the amount of network transfer. Going a step further, we can utilize the HTML5 web storage functionality to store the server response. With this technique, it is only necessary to request data from the server the first time a user visits the page. If the user returns to the page at a later time, data will be fetched from the web storage on the client side. One may question the relevance of web storage since all major browsers already have built-in support for caching. The advantage of web storage over browser caching, however, is that web storage provides much more control to the programmer.

We can store the server response in the previous section by calling the following function somewhere in the xhr.onload handler

```
function savePNGData(blob)
{
  var reader = new FileReader();
  reader.onload = function(e)
  {
      localStorage.setItem("PNGData", e.target.result);
```



Key	Value
PNGData	data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAC...
__gli	{"showRedundantCalls":true,"showDepthDiscarded":true,"ena...

Figure 3: Using Chrome's developer tools to inspect the web storage.

```
  };
  reader.readAsDataURL(blob);
};
```

This will store the image in a slot called "PNGData" in localStorage where it will exist until it is explicitly removed. An alternative to localStorage is to use sessionStorage which will be cleared when the browser session ends. The content of the web storage can be listed, modified, and deleted e.g. in Chrome's developer tools as shown in Fig. 3.

The image data stored in localStorage can be loaded with the following code

```
if ( localStorage.PNGData )
{
    var img = document.createElement("img");
    img.onload = function(e) {
        /* Convert image to vertex buffer as in the previous example */
    };
    img.src = localStorage.PNGData;
}
else
{
    /* Get the image from the server as in the previous example */
}
```

One disadvantage about web storage is that the storage limit is not guaranteed by any specification. Currently, a 5 MB limit per domain seems to be standard.

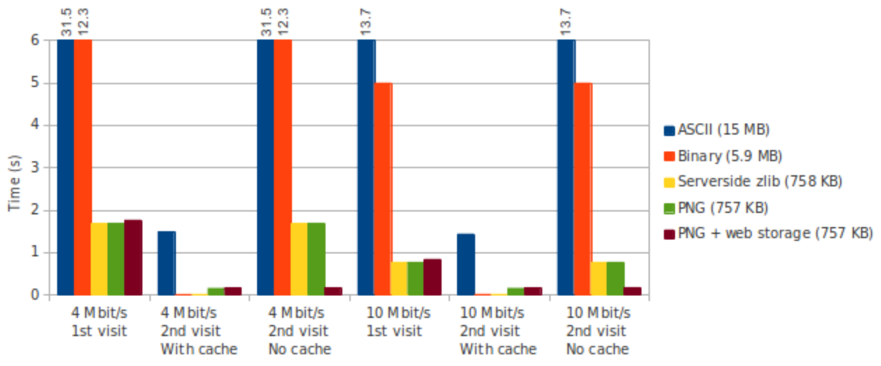


Figure 4: Loading times for the vertexbuffer used in Fig. 2. The test setup used Chrome 20 on an Intel Xeon E5620 2.4 GHz Quad Core CPU running linux.

7 Benchmark

Figure 4 shows the loading times for the vertex data used for the map shown in Fig. 2 using the various techniques described in this paper. We have limited the upload speed from the webserver to 4 Mbit/s and 10 Mbit/s. The test is available at our website [2]. We see that compression methods efficiently reduce the loading time by a factor of six to seven compared to raw binary data. A two seconds timeframe has been identified as the tolerable threshold for web page loading time for the average online shopper [1]. Hence, applying a compression scheme is essential for maintaining the audience in our case if we assume that 10 Mbit/s is a typical bandwidth for our visitors. Browser caching effectively eliminates the loading time for returning visitors, however, if the browser cache is cleared or disabled, web storage still provides almost immediate response. We see that the conversion time from PNG image to vertex data is less than 200 ms which is fully acceptable for a smooth browsing experience.

8 Summary

In this paper we demonstrated how one can optimize the loading time for large amounts of vertex data used in WebGL applications. We showed that the loading time often is limited by the speed of the network connection. Thus, using data compression such as serverside zlib compression or data encoding in a PNG image can lead to significantly increased performance. Finally, we showed how to use web storage to cache data between browser sessions.

A Binary data to PNG conversion

The following C code uses libgd [7] to create a PNG image from a binary data file.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gd.h"

int main (int argc, char** argv)
{
  gdImagePtr im;
  FILE *fh;
  unsigned char *data;
  long datasize;
  int i, j, k = 0, width = 1024, height;

  if (argc != 2)
  {
    fprintf (stderr, "Usage: %s filename.bin\n", argv[0]);
    exit (1);
  }

  fh = fopen(argv[argc-1], "rb");

  fseek(fh, 0, SEEK_END);
  datasize = ftell(fh);

  height =  datasize / (3*width) + 1;
  data = malloc(3*width*height);
  memset(data, 0, 3*width*height);

  fseek(fh, 0, SEEK_SET);
  fread(data, sizeof(char), datasize, fh);
  fclose(fh);

  im = gdImageCreateTrueColor(width, height);

  for ( j = 0; j < height ; j++ )
  {
    for ( i = 0; i < width; i++ )
    {
      gdImageSetPixel(im, i, j,
        gdImageColorAllocate(im, data[3*k], data[3*k+1], data[3*k+2]));
      k++;
    }
  }

  fh = fopen("out.png","wb");
  gdImagePngEx(im, fh, 9);

  gdImageDestroy(im);
  fclose(fh);
  free(data);
}
```

The following C++ code uses ImageMagick's C++ API [8] to create a PNG image from a binary data file.

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <cstring>
#include <math.h>

#include <Magick++.h>
using namespace std;
using namespace Magick;

int main (int argc, char** argv)
{
  if (argc != 2)
  {
    cout << "Usage " << argv[0] << " filename.bin" << endl;
    return 1;
  }

  int width = 1024;

  ifstream ifs(argv[argc-1], ios::binary);

  ifs.seekg(0,ios::end);
  int datasize = ifs.tellg();
  ifs.seekg(0,ios::beg);

  int height = datasize / (3*width) + 1;
  char * imagedata = new char [3*width * height];
  memset(imagedata, 0, 3*width*height);

  ifs.read(imagedata, datasize);
  ifs.close();
  InitializeMagick(NULL);
  Image im(width, height, "RGB", Magick::CharPixel, imagedata);
  im.write("out.png");

  delete [] imagedata;

}
```

References

[1] http://www.akamai.com/html/about/press/releases/2009/press_091409.html.

[2] http://daimi.au.dk/~thomaskj/tutorials/WebGL-VBODemo/.

[3] https://developer.mozilla.org/en-US/docs/JavaScript_typed_arrays/DataView.

[4] http://cg.alexandra.dk/2012/10/12/interactive-infographics-in-webgl/.

[5] http://httpd.apache.org/docs/2.2/mod/mod_deflate.html.

[6] http://www.html5rocks.com/en/tutorials/file/xhr2/.

[7] http://libgd.org/.

[8] http://www.imagemagick.org/Magick++/.