Web-based data visualization of an MMO virtual regatta using a virtual globe

Gerard Llorach, Javi Agenjo, Alun Evans, Josep Blat
Interactive Technologies Group, Universitat Pompeu Fabra
Barcelona, Spain
{gerard.llorach, javi.agenjo, alun.evans, josep.blat}@upf.edu

Abstract

In this paper we present the methods and techniques used to visualize the trajectory of the participants of a massive virtual regatta using a virtual globe in the web browser. The emergence of new web technologies, such as HTML5 and WebGL, have opened new avenues for visualizing and sharing 3D data. However, web-based visualization of big data is still challenging, as the power of the web browser for 3D visualization has still not reached the level of desktop applications. In this work, we use WebGL to visualize the path of the 17000 virtual boats that participated in the MMO game of the Barcelona World Race 2015, and present optimization strategies for the rendering of this Big Data (which is otherwise impossible to render in a web context on standard consumer hardware). We also combine this optimization with a render-to-texture approach to visualize the density of the boat routes, rendering and visualizing the data progressively, and using web workers for processing and managing the data.

CR Categories: I.3.7 [Computer Graphics]: Applications; I.3.3 [Computer Graphics]: Picture/Image Generation — Bitmap and framebuffer operations.

Keywords: Web 3D, WebGL, rendering, big data, optimization, visualization

1 Introduction

The Barcelona World Race is one of the world's leading sailing regattas. Crews of two people leave Barcelona and compete to sail around the world in the shortest amount of time, not stopping on land until they return to Barcelona, some three months later. Running in parallel to the regatta a web-based Massive Multiplayer Online (MMO) game allows members of the public to sail a 'virtual boat' around the world, competing in real-time with the real boats of the regatta, and navigating their boat under the same weather conditions (updated in the game every 6 hours from live weather data).

The number of players in the game provides a fascinating source of big data for visualization in a 3D application, as the trajectory taken by each virtual boat, along with the current sail choice and current weather conditions, are stored for the duration of the entire race. Using the web for visualization of big data is now very attractive due to its ease of sharing visualizations and interactivity. Although expensive and advanced renderings have still to rely on desktop applications or remote machines, the evolution of web technologies has created a great potential for web-based visualizations on the client side.

In the majority of situations, when a high quality render is required, server-side rendering is chosen and the result is transmitted as an image to the client. The main advantages of rendering on the server side are to overcome the rendering limitations on the client, and avoid transmission and parsing hold-ups. It has a negative effect on the interactivity, as user actions need to be sent to the server and the interaction can be affected by latency. Rendering the data on the client side offers a better experience for interaction, bearing in mind that the application runs on the client machine, which therefore needs to be powerful enough to run it. Also, the data needs to be transmitted and processed in an efficient and intelligent way, for the user to be able to start the interaction with minimal delay. Rendering on the client side is preferred when it is based on data and the user interacts with the visualization. Interactivity is much improved in these cases.

The main contribution of our work is a strategy for processing and visualization of complicated data for efficient rendering in a web-browser. We visualize the itinerary of 17000 boats sailing over the world's oceans, featuring over 20 million points in total. Our main problem comes from the fact that consumer level hardware is not able to render so many lines at a high frame rate, within a web context. We solve the problem by cutting the paths dynamically while simulating the evolution of the race. However, this solution doesn't permit the visualization of the all the complete paths, so we further draw all full paths into a density map progressively. Our work shows some of the possibilities of the browser for visualizing large data sets of trajectories on the client.

2 Previous work

The recommended approach for visualizing data recorded over the earth is to use a 2D map or 3D representation of the earth. We choose to use a 3D representation of the globe as most 2D earth maps have distortions; the most common 2D map projection is the equirectangular projection which distorts the poles mostly [Snyder 1987].

3D representations of the globe, also known as virtual globes, have been widely used to visualize cartographic information, weather forecasts and any other kind of information (demography, natality, GDP and many others); Blower et al. [2007] describe the use of virtual globes in three desktop applications (Google Earth, Nasa world Wind and ArcGIS Explorer). Some of these applications are very powerful, nevertheless users need to download and install each individual application, (unlike a web approach, which requires a single installation of a browser)..

Figure 1: *Render to a texture results. Texture resolutions and frame rates, from left to right: 1024 res, ~50 fps; 2048 res, ~43fps; 4096 res, ~35 fps.*

There are different approaches for implementing virtual globes on the web. For example, the applications by Gede [2009], Beccario [2014] and Prentice [2015] use all different techniques. The first uses VRML, a descriptive file format for a 3D scene, the second SVG, a XML-based file format for two dimensional vector graphics, and the last one WebGL (see Evans et al. [2014] for further information on all these methodologies). We use a custom WebGL engine because its flexibility, low-level access to the GPU and performance [Kee et al. 2012].

3 General Approach

In our work the data is presented and drawn on top of a virtual globe. Virtual globes allow the user to navigate, interact and focus in any area of the globe, and are useful for visualizing information on the earth from different perspectives and scales, allowing the display of large datasets on the globe [Beccario 2014, Blower 2007].

Our goal is to visualize and understand trajectories of the boats of the MMO game. Rendering so many lines can be difficult for consumer level hardware in terms of GPU power. We propose two different approaches to visualize the data: 1) showing the evolution of the race over time whilst only rendering a part of the trajectory of each boat and 2) rendering the full trajectories progressively into a framebuffer, resulting in a density map of the followed trajectories (figures 2 and 1 respectively). By combining approaches (1) and (2) at the same time, we achieve an effect very similar to that of rendering all the lines at once, but at a fraction of the computation time.

Having a 30 - 60 Hz frame rate is very important for the visualization display, especially for interactivity. In our work there are some calculations done in the CPU that could make the frame rate drop. To avoid this problem, we use a web worker thread. Threads in javascript don't have access to the main thread memory and the HTML document. They run asynchronously and communicate via messages with the main thread. Further details are in section Threading and Efficiency.

For storage purposes, during the race samples are only added to the dataset when the boat changes its direction or significantly changes its velocity. This creates a data set unevenly sampled in the timeline. There are approximately 1200 samples per boat on average, but each boat will have a very different number of samples according to its activity due to the sampling process. Each sample contains latitude, longitude and a timestamp. The file is stored in a binary file which can be sent to the visualization client and processed. Boat paths are stored in a binary file format where longitude and latitude are normalized to two UNSIGNED SHORT values (so we have a 2^16 different values for longitude and latitude). The position of each boat is thus stored using 4 bytes. We store an initial timestamp using 8 bytes and the remaining timestamps are stored using delta-times of 4 bytes. The total amount per boat per sample is 8 bytes during loading. To improve loading performance, this file should be subdivided and preprocessed in a previous offline stage, but this is beyond the scope of this paper.

4 Implementation details

We used two different methods to visualize the data with a custom WebGL engine. The first method consists of rendering a short path for each boat. There are too many samples to be rendered all at the same time, thus only the last samples where each boat has gone through are chosen. A mesh containing all these selected samples is recalculated as the visualization progresses. We permit dynamic changing of the number of samples for the paths, so the application can adapt to different hardware, in terms of CPU and GPU power. The second method relies on rendering the path on a texture i.e. using a framebuffer object. The result is a density map of the areas where users passed more often. The creation of this map is done in real-time while the simulation progresses. When using this method there is an important compromise between resolution and performance. High texture resolutions, which imply low performance, are required for good visualizations at a detailed scale, as a virtual globe is used for visualization and the user can zoom in specific areas. Figure 1 shows the differences when using different resolutions. The path drawn into the texture is finer and more information is appreciated, at the cost of decreasing the frame rate. The creation of this map could also be done in an offline render to achieve high resolutions and detail.

These two methods are described in detail in the following sections.

4.1 Path visualization

As mentioned before, the evolution of the race is visualized over time. Each boat has a short path or tail behind showing its last movements. This tail is represented as a line, which is formed by a specific number of samples. Each sample contains latitude, longitude and a timestamp. These samples are stored as vertices in a unique mesh that contains all the tails. As the samples are

unevenly sampled, we need to interpolate to find the position of a boat according to the progress (t) (figure 3). We tested two different approaches:

• *Interpolation through shader*: in the GPU, the vertex shader interpolates timestamps between vertices. In the fragment shader, the pixel is discarded if the pixel timestamp is bigger than *t*. The progress (*t*) is sent as a uniform to the shader.

• *Interpolation through CPU*: to find the interpolated sample at a precise time *t*, we do a binary search to find the closest sample over *t* and the closest under *t*. The new interpolated sample uses the timestamp of these last two to calculate the interpolated latitude and longitude.

When using the interpolation through shader, we accomplish a much smoother visualization between samples. This is due the fact that the progress (*t*) is sent to the shader at every update and the recalculation of the mesh can take more than one update frame. Interpolating through the GPU takes less time than the recalculation of the mesh in the CPU, which implies that it is the preferred option; in practice, however, it suffers from some performance issues. In some occasions inside the data set, vertices can be sampled very close to each other. If, while interpolating the mesh in the GPU, it does not yet contain the new vertex with a bigger timestamp (tstmp > *t*), there can be a noticeable (and undesirable) intensity change in the path drawn. To solve this, extra vertices can be added in front of the path to smooth the interpolation, but adding front vertices makes the mesh bigger and decreases performance. For example, adding three extra front vertices for each boat can result up to 51000 additional vertices.

We use another strategy to be able to visualize longer paths with the same number of vertices. We permit the application to skip a selectable number of vertices between samples when recalculating the mesh. The visualization is not completely reliable but the application is able to show longer approximate trajectories while maintaining the frame rate. Figure 2 shows the potential of this strategy: screenshots 1 and 2 are very similar although the first one is using 15 vertices per boat and the second only 4. The size of the mesh is divided by 11.

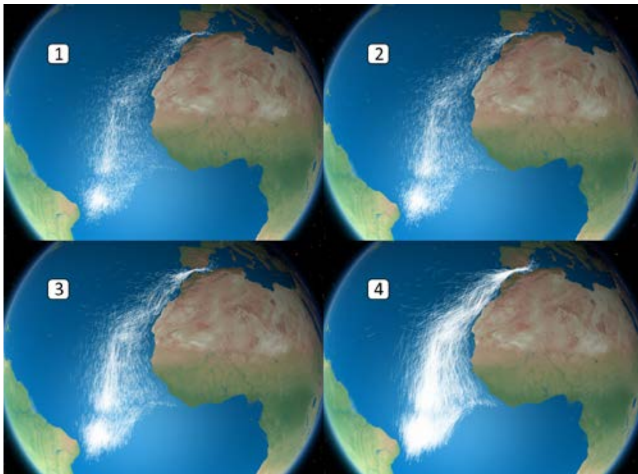

Figure 2. *Screen captures of the visualization of the paths at the same progress. Screenshot 1: 15 vertices per boat, no vertex skipping. Screenshot 2: 4 vertices per boat, 4 vertices skipped. Screenshot 3: 15 vertices per boat, 1 vertex skipped. Screenshot 4: 15 vertices per boat, 4 vertices skipped.*

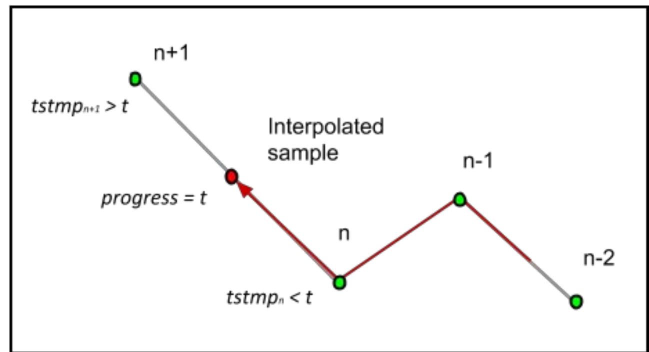

Figure 3: *Illustrative picture of the visualization of the path tails. The samples from the data set are represented by green dots (n+1, n, n-1…). The interpolated sample (red dot) is the precise position at the precise progress (t) of the simulation. The variables tstmp represent the timestamp of the near sample.*

4.2 Density map

Rendering to texture is commonly used in computer graphics for reflections and other effects. Instead of rendering to the screen, the rendering is done on a texture which can be used later on in future frames. As our data is too big to be rendered at once, we render it progressively in a texture. The result is a density map, showing the routes and areas more passed by. This density map can be created at runtime, showing the progress to the client, or it can be created previously in the server and then sent to the client. Sadly the specification of WebGL 1.0 doesn't allow the use of multisampling inside framebuffer objects (there is a draft proposal for WebGL 2.0) which would raster better quality antialiased lines. However, this could be achieved using post-processing techniques on the texture, once all the samples had been painted, with algorithms like FXAA.

4.3 Threading and efficiency

The CPU recalculation of points is usually slow, causing the frame rate to drop and to decrease the interactivity response. In our work, the process of recalculating the mesh is done in a web worker or thread. This thread is in charge of loading and parsing the data at the start of the application, as well as recalculating the mesh. The communication between the main thread and the web worker is done through petition and response. Once the last recalculated vertices are received, the main thread asks for a new recalculation. A buffer containing the new vertices is sent to the main thread, which updates the vertices of the mesh. We used typed arrays to improve performance during this process. We tested using transferable objects, a new implementation to send data faster between the thread and the main thread. We discarded using them because transferable objects meant creating new array buffers at every mesh update, which decreased significantly the performance of the visualization.

5 Results

Several methods were applied to accomplish the visualization of the trajectory of a large number of boats in a virtual regatta around the globe. Rendering to a texture permitted an analysis of the areas and paths most followed by the players. The density map can be useful to understand where the players thought the winds would be better to win the race and, which routes they preferred.

The path visualization shows precisely the trajectory of each individual boat over time, which cannot be shown with the density map alone.

For the path visualization we choose to use the interpolation through the CPU. This is due the fact that the interpolation through shader can cause some undesired intensity changes in the overall path visualization, as mentioned above. Nevertheless, in our application we allowed the users to choose a particular number of boats and load all their samples in a mesh, so they can visualize precisely the whole trajectory of the selected boats. For this visualization we created a mesh containing all the points of the selected boats from the beginning to the end of the race. As this mesh doesn't need to be updated the interpolation through shader is the best choice.

When rendering on the client, we have to bear in mind that every client will have a different machine. Allowing users to choose the number of vertices per path and the number of vertices to be skipped makes the application adaptable to the client's machine. The visualization might be slightly different, but it can still provide very similar results. Table 1 provides evidence that visualizing all the information without using our techniques is not possible. Table 2 shows the performance of the application with different configurations. Combining both techniques, the path visualization and the density map, allows the visualization of the trajectory and the density evolution at the same time. The numbers shown in these tables are partially illustrative, as they were all obtained from the same machine (Windows 8 x64 2.50 GHz, NVIDIA GeForce GT 750M).

Table 1. *Results without using the proposed techniques.*

Nº of full paths	Nº of vertices	Frame rate (fps)
500	614.915	45
1.000	1.235.883	38
5.000	6.171.165	14
17.000	20.292.474	App crashes

Table 2. *Results using the proposed techniques with 17.000 boats. The last results show the frame rate with both approaches (path visualization and density map) used at the same time.*

Nº of vertices per boat	Nº of vertices	Update mesh rate (fps)	Texture resolution	Frame rate (fps)
8	136.000	14.9	-	56
15	255.000	11.2	-	49
30	510.000	6.1	-	42
15	255.000	12.3	1024	48
15	255.000	11.1	2048	41
15	255.000	10.3	4096	28
15	255.000	8.8	8192	14

6 Future work

This paper has proposed some strategies to achieve the visualization of an MMO virtual regatta. Nevertheless, some other techniques were not used and should be mentioned for future work.

· To reduce the size of the recalculated mesh containing the tails of the boats, the viewing frustum could be considered to discard samples that fall outside. As the mesh is continuously updated this would not suppose a big change in the structure of the code.

· Using more threads to recalculate the mesh would improve the update rate of the mesh with the short paths. However, this parallelization would depend on the number of cores of the machine and it would not solve GPU related bottlenecks.

· When rendering the density map we tried to visualize the density of the areas where most boats went by. For this we used a very low alpha value for each boat's path, which meant that paths that would not fall on the main trajectory would be almost invisible. If alpha is set to have a bigger value, the result in busy areas will show the same density. One solution would be to use high dynamic range techniques to obtain information about the density even when the pixel's resultant alpha value is over 1. Another solution would be to change the alpha value of each path, or part of the path, according to its contribution in the result, i.e. density map.

· One of the perks of rendering on the client is that the data needs to be transmitted first. In our case the file size is approximately 150 MB, which can take a long time to download. Parsing all the data is also time consuming and does not allow the simulation to run until all the data is parsed. Preprocessing and splitting the binary file into smaller temporal subdivisions would allow the simulation to start without long hold-ups.

· Another advantage about virtual globes is that information can be added on it, like the weather conditions and ocean currents. Wind forecasts during the evolution of the race were added to the application, but they are out of the scope of this paper.

All the techniques shown in this paper are very particular for this case, but they could be extended for many other applications. For example, showing the trajectory and paths of the inhabitants of a city, bird migrations and many others.

Acknowledgments

This work has been partially funded by the Spanish Ministry of Science and Innovation (TIN2011-28308-C03-03).

References

BECCARIO, C. 2014. Earth Wind Map, URL: http://earth.nullschool.net, last accessed: March 2015

BLOWER, J. D., GEMMELL, A., HAINES, K., KIRSCH, P., CUNNINGHAM, N., FLEMING, A. AND LOWRY, R. (2007) Sharing and visualizing environmental data using Virtual Globes. In: UK e-Science All Hands Meeting 2007, 10-13 September 2007, Nottingham, UK, pp. 102-109

EVANS, A., ROMEO, M., BAHREHMAND, A., AGENJO, J., BLAT, J. 2014. 3D Graphics on the Web: a Survey. Computer & Graphics, vol 41, pp 43-61. ISSN 0097-8493

GEDE, M. 2009. Publishing Globes on the Internet. Acta Geodaetica et Geophysica Hungarica, vol 44, pp. 131-148

KEE, D.E., SALOWITZ, L., CHANG, R. 2012. Comparing Interactive Web-Based Visualization Rendering Techniques. Tufts University, Medford, MA.

PRENTICE, C. 2015. Iss Photo Viewer, URL: http://callumprentice.github.io/apps/iss_photo_viewer/index.html, last accessed: March 2015

SNYDER, J. P. 1987. Map Projections – A Working Manual. U.S. Geological Survey Professional Paper 1395, Geological Survey (U.S.)