Web-Based Mobile Display of 3D Models

Elijah Verdoorn, Joe Peterson, Austin Pejovich
Department of Mathematics, Statistics, and Computer Sciences
St. Olaf College
1500 St. Olaf Avenue, Northfield MN, 55057
verdoo1@stolaf.edu, peters10@stolaf.edu, pejovich@stolaf.edu

Abstract

With the rapidly increasing processing power available to mobile devices, client side rendering is becoming a more and more viable option for mobile rendering. Recent developments in mobile graphics as well as research done in the field of web-based applications leads to the conclusion that a client-rendered model is increasingly possible on consumer grade handhelds. It is the purpose of this paper to explore and qualify the extent to which client-side rendering of a 3D model is a reasonable venture, and to test the capabilities and limitations therein. Testing of these capabilities requires optimization techniques on both the client and server. Rapid page load was achieved through use of pre encoding of data on a server, then this page load data is compared to the WiFi signal strength and the network bandwidth data that was collected throughout a large building.

Introduction

Historically, the practice of rendering 3D graphics over the internet for mobile devices was a process of trying to most efficiently send the already rendered data over the network from server to client. However, with the rapidly increasing processing power available to mobile devices, this is no longer the only answer. Recent developments in mobile graphics as well as research done in the field of web-based applications leads to the conclusion that a client-rendered model is increasingly possible on consumer grade handhelds. The purpose of this paper is to explore and qualify the extent to which client-side rendering of a 3D model is a reasonable venture, and to test the capabilities and limitations therein.

Background

Server-side rendering tends to handle the obvious bottleneck of data transfer by trying to predict exactly how much information it needs to send at once or in simplifying that information as much as possible. One 2007 study worked to be able to give the smartphones of the day the ability to render textured polygons in the order of millions by having a cluster of specialized PCs rendering the information and sending the render as a MPEG video to be streamed by the client [Sawicki and Chaber 2012].

Valuable bits of bandwidth can be saved by relying on client-side hardware to do the rendering of a 3D model and mobile devices are getting much better at performing these graphical computations. The question of efficiency then becomes a question of how to send the information in a way that is quickly readable by client-side hardware and smoothly sent over the network. One such method is to use predictive modelling. A 2012 paper from the *Journal of Computer Science* detailed an approach to this method [Lamberti and Sanna 2007]. In this paper, the authors take a 3D model and divide it into 3D scenes, each with their own objects and 3D meshes. they provide the starting scene to the client, and then constantly stream information about the surrounding scenes and meshes. This works because the only information sent–both initially and in the stream–is the parts of each scene or mesh that are visible. This means that even though an object might be in the frame of the camera, only the triangles in the mesh that are not occluded are sent to the client. This method relies not only on the hardware of the client, but on the hardware capabilities of the server. The server is constantly surveying the 3D model and trying to predict which scenes and meshes will become visible to the camera view of the client. While hardware intensive, this method heavily reduces strain on the network, and increases responsiveness of the mobile display. In addition to rendering the information being sent by the server, the client builds up a cache of information while the rendering is happening, and deallocates some information at a certain interval based on the same predictive model that the server uses to guess where the user is navigating in the 3D model. The cache can be used to recall data without relying on the server to resend that data.

Additional work done in this area comes from the work of Sawicki and Chaber, who attempt to answer a similar question.[Sawicki and Chaber 2012] Their paper, however, uses the

older and less efficient canvas renderer, while the data presented in our research is not only using modern devices, but also the more efficient WebGL renderer, which allows for hardware acceleration. Their claim (published in 2012) that "it is currently not possible to rely on WebGL technology" has changed with the passing of time, currently 91% of browsers are capable of supporting the WebGL renderer used in this paper. [WebGLStats 2015] Additional work has gone into considering texturing the meshes that we use, a facet of 3D models not considered in Sawicki and Chaber's paper.

The problem of web-based 3D graphics is driven by convenience. Today's users are accustomed to near-instant response when interacting with the digital world, especially the brief yet numerous interactions had daily with mobile technology. This applies to 3D models as well: reactivity and interactivity are two of the five most important qualities that any computer visualization can have [Tang et al. 2010]. The other qualities focus on other aspects of the experience, which are beyond the scope of this paper. With advances in hardware capabilities of mobile devices it is becoming easier to provide this experience to users, even when there is a large amount of data involved. For the user, interactivity is the most important quality that an application can have, thus driving inquiry into this problem; the quality of device and circumstances such as signal quality and location ideally should not affect the user's ability to manipulate the model.

Method

Given a large database of 3D polygon data and texture mapping information, along with texture images, a system was devised for the render of these polygons. These polygons are an attempt to create a 3D representation of a scene contained in carefully measured stereo images. We concern ourselves with the creation of a method of displaying a model that is reasonably responsive regardless of the number of simultaneous users of devices connecting, or the bandwidth and hardware available to any one device.

Server

The server involves being able to easily retrieve information about the 3D scene and images being used for textures in a way that can be sent over the network using Javascript Object Notation, or JSON. We chose to use JSON for our communication standard due to it's ease of use and cross-language compatibility. The JSON structure is extremely versatile and has no limit to the size or function of the information being transmitted. Our JSON structure sends the name of a 3D object, the name of an associated image (from which to pull a texture), a set of coordinates that describe vertices of polygons to be rendered, and (U,V) coordinates associated with the vertex. The JSON object also includes the image data that polygon textures are derived from. Encoding the original .png images in base64, we can send the data as a part of the JSON object. This is not the most efficient method, as the effective filesize of the image is increased to 4/3 the original size, but it is the most straightforward given our dataset. In addition to encoding

the data, the server also checks to make sure tessellation data is up to date. Because the raw data is stored in a structure that is not easily renderable, the server converts that data into a set of renderable points, which are stored in a file so that time is not wasted recreating this data for every request received. Files created for this purpose are known as .poly3GL files, and hold information about which images that particular polygon exists in, and then 5 coordinates for each vertex, grouped to contiguous polygons.

We take advantage of the similarities between the desktop library OpenGL and the web library WebGL to precompute as much as possible. OpenGL is used to take patches of vertices and convert them to GL primitives so that the entire polygon is divided into triangles that can easily be rendered and texture mapped. For simplicity we use only a selection of the available primitive types, namely the triangle, triangle strip, and triangle fan [Figure 3]. This form means that we can save time parsing by packaging information by polygon rather than by image pair, so even though the text files are longer, the information is more centralized and more easily read into the 3D model by the client.

The core of the server is implemented as an extension of the Python standard HTTP server. By extending this widely-used server we can take advantage of the work done on optimising the central server functionality and focus on efficiency in programming the specific functions that we need. We implement a variety of functions in Python, but the computationally intensive task of creating .poly3GL files is handled in a C++ program to ensure that it is as efficient as possible. Our server attempts to prevent this from having to be done on the fly by creating these files periodically for any image that has been edited, but occasionally a user will attempt to render information that has not been updated. In this case the computation has to be done before the server's response can be generated. This cannot be avoided if we expect the scene to make sense as any changes to one file could cascade into changes to the entire scene.

Client

Once the polygon information is received by the client, it is the job of the client hardware to efficiently render and control the 3D model. The first and simplest thing the client does is condition a render distance. This is relatively easy to implement because there is already a loop iterating through each value in the JSON object. It is therefore relatively inexpensive to add conditional logic to render polygons based on distance. The client could also skip rendering polygons which are partly or completely occluded by other polygons, as the 2012 paper on predictive modeling does [Lamberti and Sanna 2007]. This is not something implemented in this version of the 3D model because the information being grabbed by the server is not yet accurately separated into scenes and object meshes like the data used in that study was.

To avoid socket programming, the implementation of the client-server communication uses AJAX. AJAX is a method of communication that takes advantage of asynchronous programming. For our purposes we chose to use POST requests within our AJAX communication system. The client assembles a set of headers that the server needs to recognize the request as an AJAX request, then encodes data to the request. The client sends the data to the

server on an open port, determined when starting the server. The server receives the request and recognizes it as AJAX based on the headers that the client generated. The server can parse and use the data that the client encoded, make decisions based on said data, then encode its own reply, also formatted with headers and data. After the client sends the request, it can continue doing something else while it waits for a response; this property is especially important for our program since any time spent waiting for the server to respond is time that the client cannot spend rendering frames for the user. The reception of the server's response triggers a client callback, the callback triggers a routine that adds the contents of the server's JSON encoded response to the rendered scene, which is then rendered on the next frame.

The server's response is a JSON object. This object is parsed, then the points that it contains are rendered in 3D space. To ease this process we chose to use the Three.js library, which is a lightweight library on top of WebGL. Having defined a set of primitives for Three.js to use, we feed the vertex data into the scene one face at a time. After the vertices are all in place, the base64 image data contained in the JSON object is assembled into a Three.js image object. By going directly into an image object we can reduce the amount of time that the client must spend reassembling the image. The use of base64 is further justified when the asynchronous requests are considered, as the only delay that the larger filesize causes is in the initial load. The U and V coordinates are then used for each vertex to find the appropriate coordinate on the 2D image. Once the coordinates are mapped for the entire face the texture is applied to the face and it is added to the scene. This entire process only happens once for each set of data that the client receives, the rest of the time the scene is simply redrawn based on the position of the camera.

Once the data is rendered by the client the user can begin to interact with the model. The versatility of web applications is showcased here, as the program changes the interaction controls based on the hardware that a user is connecting from. If the device reports that it is a mobile device then the client allows for motion-based controls where rotating the physical device rotates the camera and touching the screen moves the camera about the scene; if a device reports to be a traditional computer device (laptops included) then the client allows for more keyboard friendly controls. This is certainly possible in a more traditional application, but the web app can reuse the majority of it's code regardless of device. Additionally, this method of programming made testing the program much easier.

Data

The mobile devices used for testing were an Apple iPhone 5, Samsung Note 4, an Asus ZenFone, a first generation Apple iPad Air and an HTC Nexus 9. By using a varied set of devices we were able to gather data that simulates a variety of use cases. Also tested were an HP z420 Workstation desktop PC (from which the servers were run) and a Lenovo N585 laptop. [Figure 2] All tests were performed in the Google Chrome browser or the Chromium browser, from which Google Chrome is derived. This browser was chosen for its JavaScript engine, which is known to be the most efficient on mobile devices. Because of possible variation in how different browsers might handle our code, we chose to use chrome to ensure the data did not have confounding variables, and the version of Chrome was noted for each device.

Each device was tested in four different locations, with varying connection strengths to a WiFi network. For comparison, the network upload and download bandwidth according to www.speedtest.net was recorded. The loading times and the average frame rate over the first minute were recorded for a variety of rendering situations ranging from a single polygon to all 422 polygon faces available from the database. As expected, the frame rate and the loading times were directly affected by the hardware used to run the program. The HTC Nexus 9 tablet consistently came out leading the pack, rendering all but the most complicated scenes with frame rates over 30 frames per second and loading times on the low end of average. On the other end of the spectrum the Apple iPhone 5 and iPad were unable to complete all the tests, failing when the model was presented with too many polygons. When these devices did function, they were sluggish and unresponsive, barely managing ten frames per second on the 422 polygons test.

The variance in location, meant to allow for comparison of data on various connection strengths, quickly revealed that the network connection speed, so long as it remains above a modest 20 Mbps, is not a significant factor in the rendering of a model. As of March 15, 2015 the average internet speed in the United States of America is 33.9Mbps according to Ookla, the company behind www.speedtest.net (used in our tests). [Average… 2015] This being the case, our research became easier as we did not have to put as heavy an emphasis on the connection speed data, and revealed to us that the vast majority of the latency that we experience when rendering our model is the fault of the code, not the latency caused by an internet connection.

After noting these results and recognizing that users of mobile devices expect speed, it was proposed that another rendering scheme be devised, resulting in an implementation of a dynamic rendering method. additional tests were performed using a dynamic rendering method, starting at 36 polygons and ending with 103 polygons. The same data was recorded, although it should be noted that the loading time data in this case represents the time that a user would have to wait from loading the page to the time that they could begin to interact with the model. Portions of the model are rendered based on the user interaction. During the test, a new set of data was rendered when the user moved a distance of 50 units. Page load time significantly improved when comparing this new data with data gathered when rendering all of the images at once. Dynamic rendering method improved page load times by 64.66% across all devices over sending the data in one large package. The HTC Nexus 9 saw the greatest improvement, moving from an average of 1016.5 ms to 297 ms, a 70.78% decrease. These changes produced an average decrease in frames per second of 28.97%. This decline can be accounted for in the processing: each batch of data is added to the scene directly after it is received. This means that while the dynamic rendering solution will reduce page load time, each batch of data must be processed as they arrive, cutting into the computational cycles that can be dedicated to rendering frames. In the end we are using the same number of cycles to add the new portions of the geometry to the scene, simply shifting when they occur.

A system was implemented to preload the data needed to be sent in the initial JSON object so that no time is wasted between the AJAX request being sent and the JSON being received by the client. This was implemented with the hope of driving page load time down even further, and the data gave surprising insight into how the page load time can be broken down into

the different stages of communication between the server and the client and the client rendering the data. To establish a baseline, the preemptive load functionality was implemented in the single polygon test used earlier. This version of the server effectively provides information about how quickly it takes the server and client to communicate with near empty data. The average page load on the original single polygon test was 343.9 ms. After changing to the preemptive loading version of the server, average page load time decreased to 322.6 ms. Additionally, one should remember that there is unavoidable variance in communication time caused by radio interference, unrelated network traffic, and communication bandwidth being used by the device for background processes, all of which contribute to the range of speeds that we experienced in testing, a total range of 78.35Mbps. This 6% decrease in load time was a modest efficiency gain, but could be ascribed to variance in connection strength and network noise. To follow up on the inconclusive results, further tests were implemented in the dynamic loading version of the server, as well as in the 422 face version. After a second round of tests, we found that the decrease in page load time did in fact scale with data size, with the page load time decreasing by an astounding 73% when preloading the 3D information of all 422 faces in the large test server [Figure 6]. While the idea is reminiscent of server-side rendering, it is instead simply an efficiency gain through reducing time required in reading files on the server and moving the data to memory, which is much faster than accessing the disk for every bit of data.

Conclusion

Future work would focus its efforts on optimizing the data being transmitted to the user. Of the possible optimizations, the compressing of the polygon mesh would allow for more efficient streaming. One method used by [Tang et al. 2010] is using edge collapse and vertex split. Research into this topic could also focus on the optimization of the server data structure, making a robust system that goes beyond the simple text files that were used in these trials.

The base expectation of the work was a 3D model rendering on the web. This task, previously immensely challenging, is now easily completed for basic models using a client-server relationship written using Python and JavaScript. Assorted devices rendered the model with varied success, upon seeing the results, some modest optimizations were implemented. Using dynamic rendering proved to be both quick loading and allowed for FPS averages at levels more than useable. Pre-encoding some of the data that the server would have to frequently use further increased the speed of loading a page. These simple optimization strategies pushed the program beyond satisfactory on the majority of devices, and increased usability on all platforms.

References

Ziying Tang, Xiaohu Guo, and Balakrishnan Prabhakaran. 2010. Receiver-based loss tolerance method for 3D progressive streaming. *Multimed Tools Appl Multimedia Tools and Applications* 51, 2 (2010), 779–799.

V. Vani, R.Pradeep Kumar, and Mohan S. 2012. 3D Mesh Streaming based on Predictive Modeling. *Journal of Computer Science* 8, 7 (2012), 1123–1133.

Anon. 2013. Microsoft Assigned Patent for Pipeline for Network Based Server-side 3D Image Rendering. *Targeted News Service* (February 2013).

So-Yeon Yoon, James Laffey, and Hyunjoo Oh. 2008. Understanding Usability and User Experience of Web-Based 3D Graphics Technology. *International Journal of Human-Computer Interaction* 24, 3 (2008), 288–306.

Ying Zhong and Chang Liu. 2013. A domain-oriented end-user design environment for generating interactive 3D virtual chemistry experiments. *Multimedia Tools and Applications* 72, 3 (December 2013), 2895–2924.

Fabrizio Lamberti and Andrea Sanna. 2007. A Streaming-Based Solution for Remote Visualization of 3D Graphics on Mobile Devices. *IEEE Transactions on Visualization and Computer Graphics* 13, 2 (2007), 247–260.

Bartosz Sawicki and Bartosz Chaber. 2013. Efficient visualization of 3D models by web browser. *Computing* 95, 1 (August 2013), 661–673.

Bartosz Sawicki and Bartosz Chaber. 2012. 3D Mesh Viewer Using HTML5 Technology. *Electrical Review* (2012), 155–157.

2015. WebGL Stats. (October 2001). Retrieved January 28, 2016 from http://webglstats.com/

2015. Average United States Download Speed Jumps 10Mbps in Just One Year to 33.9Mbps. (March 2015). Retrieved January 28, 2016 from http://cordcuttersnews.com/average-united-states-download-speed-jumps-10mbps-in-just-one-year-to-33-9mbps/

Appendix

Figure 1 - Terms

contour - a tile converted to 3D, the contour is a single plane in 3D space.
face - one plane of a 3D object.
image - a set of 3D data that is used to create tiles and contours, also used to texture a set of contours. Images come in pairs, one image's worth of tiles refers to all the tiles (and therefore, contours) that appear in that image.
texture - the image displayed on a face, taken from an image.
tile - a representation of a 2D planar object.

Figure 2 - Test Platform Information

Device	OS Version	Browser	Browser Version	CPU	GPU	RAM [GB]
iPad Air	iOS 9.2.1	Chrome	47.0.2526.107	Dual-core 1.3 GHz Cyclone	PowerVR G6430	1
Lenovo N585 Laptop	4.2.5-1-ARCH (Arch Linux)	Chromium	47.0.2526.106	AMD E1-1500	Gallium 0.4 on AMD PALM	4
Samsung Note 4	Android 5.0.1	Chrome	47.0.2526.83	2.7 GHz Quad-Core Processor	Adreno 420	2
Google Nexus 9	Android 6.0.1	Chrome	44.0.2403.133	Nvidia Terga Dual-core 2.3 GHz	Kepler DX1	2
ASUS Zenfone 2	Android 5.0.0	Chrome	47.0.2526.83	Intel Atom Z3560 @ 1.8GHz	PowerVR G6430	2
Workstation PC	Fedora 22	Firefox	43.0.3	Intel® Xeon(R) CPU E5-2665 0 @ 2.40GHz × 4	Gallium 0.4 on NVC3	8
iPhone 5	iOS 9.2.0	Chrome	47.0.2526.107	Apple A6 Dual-Core 1.3GHz	PowerVR SGX 543MP3	1

Figure 3 - WebGL Primitives

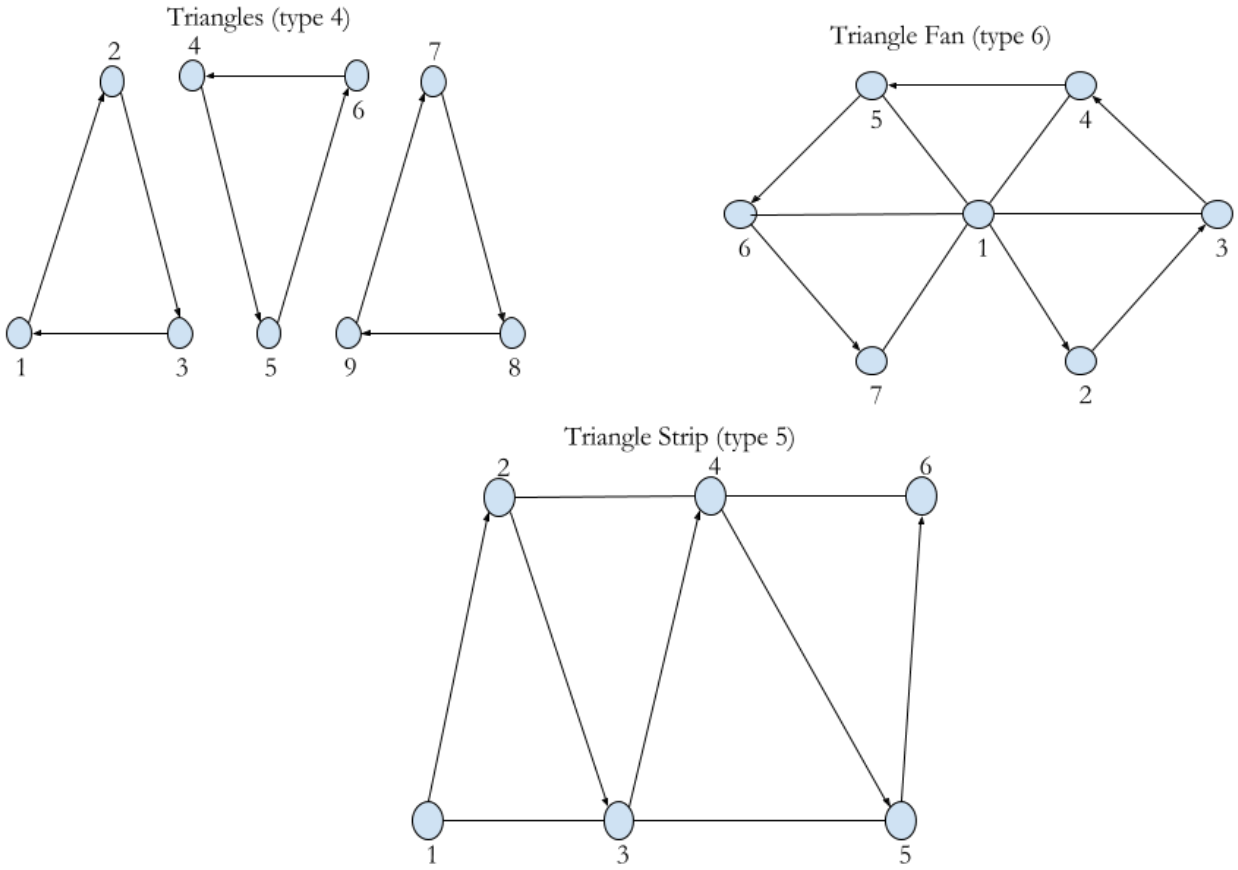
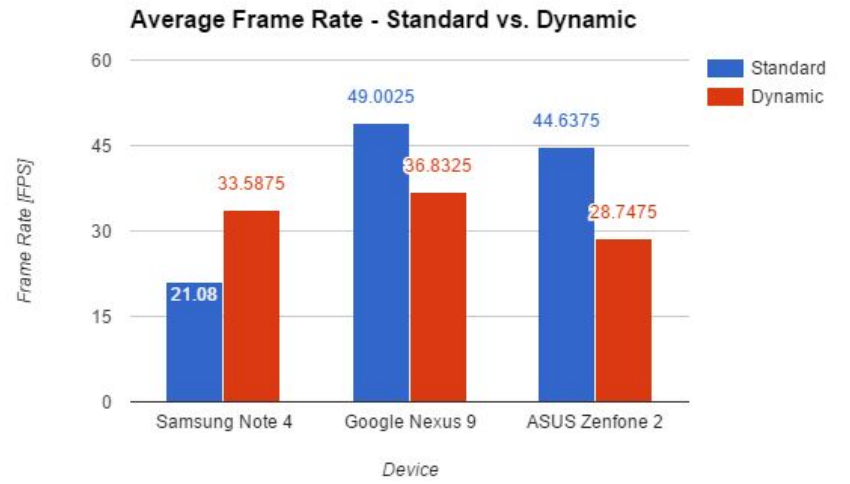


Figure 4 - Dynamic Rendering vs. Standard Loading (not pre-encoded) (103 Polygons)

Standard Method		
Device	**Average Page Load [ms]**	**Average Frame Rate [FPS]**
Samsung Note 4	1437.25	21.08
Google Nexus 9	1016.5	49.0025
ASUS Zenfone 2	1997.5	44.6375

Dynamic Method		
Device	**Average Page Load [ms]**	**Average Frame Rate [FPS]**
Samsung Note 4	634.25	33.5875
Google Nexus 9	297	36.8325

ASUS Zenfone 2	605	28.7475



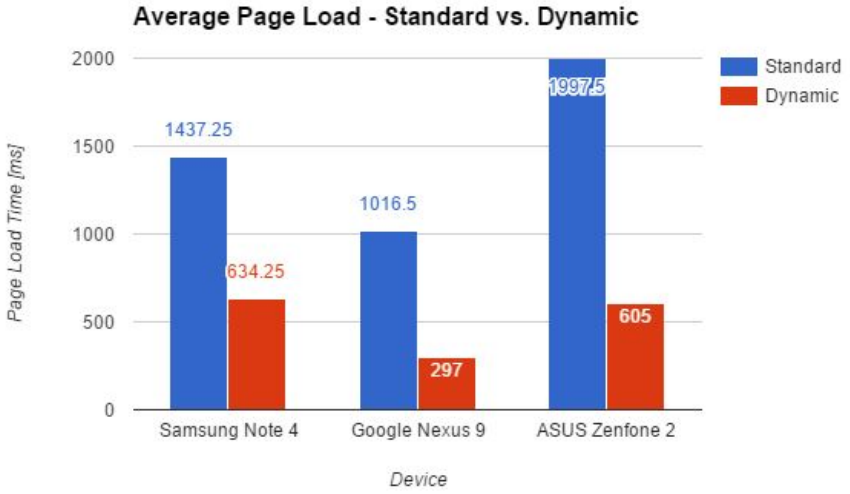


Figure 5 - No Optimization Data: 103 Polygons

Device	Average Frame Rate [FPS]	Page Load Time [ms]
iPad Air	10.425	1292.25
Lenovo N585 Laptop	10.03	914.25
Samsung Note 4	21.08	1437.25
Google Nexus 9	49.0025	1016.5
ASUS Zenfone 2	44.6375	1997.5

Workstation PC	16.01	461
iPhone 5	7.13	2699.666667

Figure 6 - No Optimization Data: 1 Polygon

Device	Average Frame Rate [FPS]	Page Load Time [ms]
iPad Air	288	55.355
Lenovo N585 Laptop	197.5	57.34
Samsung Note 4	501.25	59.3275
Google Nexus 9	217.75	59.7125
ASUS Zenfone 2	557.5	58.0125
Workstation PC	210	32.9
iPhone 5	517	58.85

Figure 6 - Page Load Time: Pre-encoding vs. Standard

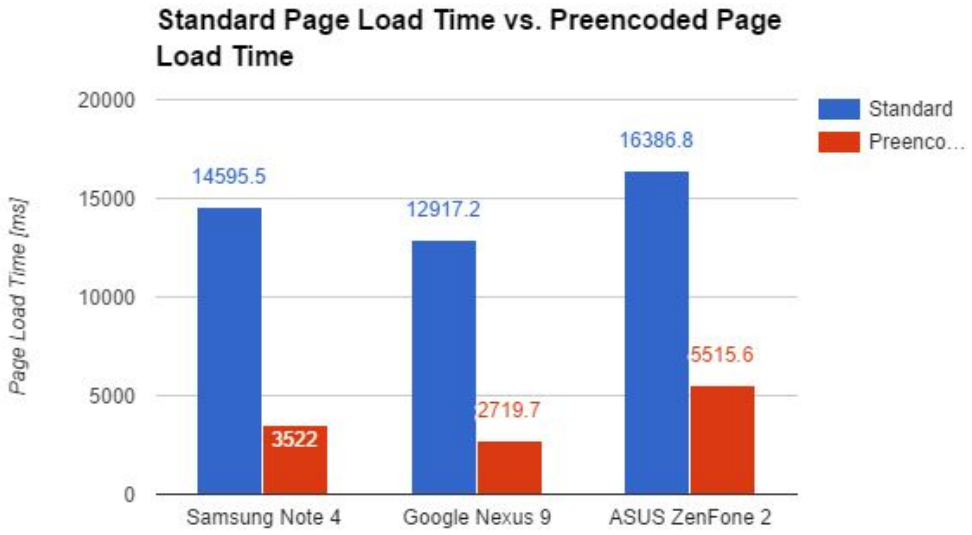


Figure 6 - No Optimization Data - 422 Polygons

Device	Average Page Load [ms]	Average FPS [ms]
iPad Air	Crash	Crash
Lenovo N585 Laptop	3204	4.77
Samsung Note 4	4664.5	8.3625
Google Nexus 9	3077.75	19.33
ASUS Zenfone 2	7227	34.8225
Workstation PC	1858	15.85
iPhone 5	11695.5	2.475