Graphics' Card Utility with WebGL and N-Buffering

Improving performance using N-buffer strategies with WebGL

Kapacitetsutnyttjande av grafikkort med WebGL och N-buffert

Förbättrande av prestanda genom N-buffert strategier med WebGL

EMANUEL PALM

Degree project in
Computer Engineering
Undergraduate, 15 ECTS points
Supervisor at KTH: Jonas Wåhslén
Examiner: Ibrahim Orhan
School of Technology and Health
TRITA STH 2014:70

KTH Royal Institute of Technology
School of Technology and Health
136 40 Handen, Sweden

Abstract

This thesis covers the utilization of N buffers in order to reduce resource contention on an abstract WebGL machine, and subsequently freeing up machine processing time. The buffers touched are frame buffers and vertex buffers.

The paper also briefly covers the purpose and function of N buffering in relation to graphics and the function of a WebGL machine, the research and production of benchmark prototypes, some relevant benchmark results, and analysis and conclusions.

The conclusion is made that the use of N>1 buffering is a potentially viable strategy for increasing WebGL performance, and some theories are outlined and suggestions given for further research to be made for the resolving of how this performance gain may be improved.

The thesis was produced in cooperation with Citerus AB.

Keywords

WebGL, graphics, OpenGL ES, performance, optimization, buffering, triple buffer.

Sammanfattning

Denna uppsats granskar nyttjandet av N buffrar för att minska uppkomsten av resurskonflikter på en abstrakt WebGL maskin, och således frigöra processtid på maskinen i fråga. De typer av buffrar som vidrörs är så kallade f*rame buffers* och v*ertex buffers*.

Uppsatsen går också igenom syftet och funktionen hos N buffrar i relation till grafik och funktionen hos en WebGL maskin, förstudien kring och produktionen av prestandatest-prototyper, en del relevanta mätresultat, samt analys och slutsatser.

Slutsatsen nås att nyttjandet av N>1 buffrar är en potentiellt gångbar strategi för ökandet av prestanda hos WebGL, och en del teorier presenteras och förslag ges kring ytterligare studier för att öka prestanda ytterligare.

Examensarbetet producerades i samarbete med Citerus AB.

Nyckelord

WebGL, grafik, OpenGL ES, prestanda, optimering, buffert, trippel buffert.

Preface

This paper describes my degree project in computer engineering at KTH (The Royal Institute of Technology), carried out in cooperation with Citerus AB.

In order to fully grasp the technical reasoning and specifications included in this report, some superficial knowledge of general software engineering, statistics, and JavaScript is required. A working knowledge of OpenGL/WebGL would aid a reader, but is not taken for granted.

I would like to thank Citerus AB for the opportunity to work together with them, and especially Sandra Sundberg, Frand Berglund, Magnus Olsson and Pia Björlin, with whom I spent most of my time working while at Citerus. These people taught me many things and made the degree project a very valuable experience.

Table of Contents

1. Introduction

This section outlines the purpose and ambitions of the project, the contents of the thesis, and other relevant information in relation to those subjects.

1.1 Background

The intent of this degree project in computer engineering is to investigate the advantages and disadvantages which N-buffering may yield for applications build around WebGL, for the sake of increasing the utilization of rendering capacity of some, imaginary or real, graphics machine.

Euclidian geometry applied in rendering computer graphics potentially requires significant amount of computation capacity. Describing complex shapes such as trees, rocks, roads, cars, etc. may require amounts of data beyond the limits of contemporary graphics machines. For this reason, applications rendering geometry commonly rely on dedicated hardware circuits. WebGL [1] is an open standard offering a standardized interface to this kind of hardware, through JavaScript in a web browser, which facilitates application distribution through Internet browsers.

Of paramount significance is the efficient utility of WebGL machines in relation to applications where geometry is complex, since the degree of efficiency vastly impacts the usefulness of applications. Improving WebGL efficiency also serves to improve device energy consumption, or reduce requirements on hardware complexity where battery time, budget, or other factors may be critical.

1.2 Goals

The aim and scope of this thesis is to investigate one particular strategy, namely N-buffering, with the purpose of evaluating its potential in increasing the performance utilization of an abstract WebGL machine.

The below list reflects the questions answered by the thesis, and the concrete tasks performed in order to answer those questions and thus achieve the thesis goal.

- **Theory and Research**
  - What is N-buffering and how does it affect performance?
  - Which are the more significant functions of WebGL machines?
  - When will the usage of N buffers help increase the utility of WebGL?
  - Which use cases will best reflect normative usage of a WebGL machine?
  - Which measurements most adequately reflect the performance of a WebGL machine?
- **Experimentation**
  - Decided on and implemented prototype reflecting normative use cases for WebGL.
  - Collected data from produced prototype.
- **Analysis**
  - Related any relevant statistics from the gathered data.
  - Stated the formal conclusion about the potential gains of N-buffering in WebGL.

1.3 Delimitations

The most notable delimitation is the narrowing down the scope of improving performance to only N-buffering. Graphics machines are commonly complex, and there would be much room for optimization by considering other aspects of such. However, as time, and experience within the domain, are scarce, the scope was thus restricted.

The time available and the lack of experience within the field, likely and notably lead to the search for potential synchronization issues in the abstract WebGL machine not being exhaustive, and the choice of prototypes to be implemented not taking into account issues and complexities undiscovered by the author.

1.4 Employed Methods

Firstly, research was made in relation to how an abstract WebGL machine works, and how it is utilized. As sufficient knowledge in this area was acquired, then research moved on to determining potential synchronization issues solvable through N-buffering.

Secondly, attention was directed toward finding viable strategies for implementing N-buffer systems in WebGL.

Thirdly, some common usage scenarios for WebGL were decided upon, chosen in relation to how they would be thought to burden the abstract WebGL machine. Those scenarios were materialized into a HTML/WebGL benchmarking prototype, used for simulating different categories of WebGL application.

The Mozilla Firefox [2] browser was used for debugging and testing the prototype. Firefox was used since it provides platform independence and also had JavaScript analyzation tooling. Google Chrome was also employed as a complement.

For better organizing the JavaScript code written, the require.js framework [3] was used.

1.5 Chapter Overview

For the sake of clarity and to simplify the use of the thesis, a list of the chapters is here presented.

1. **Introduction** (page 1)

   Relates the background and scope of the thesis.

2. **Current Situation** (page 5)

   States the current situation of the industry revolving around the rendering of real-time 3D graphics.

3. **N-Buffering** (page 7)

   Explains the concepts of using one, two, or three or more screen buffers between a computer application and a monitor.

4. **The Abstract WebGL Machine** (page 11)

   Outlines the stack of languages and technologies required in order to produce WebGL applications, briefly explains the WebGL rendering pipeline, and also relates the purpose and organization of the WebGL API.

5. **Research on Synchronization** (page 15)

   Relates the research made on the topic of synchronization in relation to the WebGL machine, depicts an imaginary graphics machine and points out where synchronization issues may occur. Also, relevant N-buffer strategies for overcoming these issues are presented.

6. **Performance Analysis of WebGL Machines** (page 19)

   Briefly explains how normative usage of the WebGL machine looks in relation to the research on synchronization, and lists metrics which will be used to analyze the WebGL machine.

7. **Prototype Implementation** (page 21)

   Describes the produced prototype from an implementation standpoint, and accounts for the collection of data from the prototype.

8. **Data Results and Analysis** (page 25)

   Presents significant statistics produced through the use of the prototypes.

9. **Discussion** (page 31)

   Contains a brief discussion, and some observations, on the topics of improving general WebGL performance further, and the implications of improving WebGL performance.

10. **Conclusion** (page 33)

    Accounts for the conclusions made, by the author of the thesis, in relation to the usefulness and implications of using N buffers in relation to WebGL.

2. Current Situation

A brief overview of the current situation surrounding WebGL and such technologies, in relation to the thesis.

Computational devices become increasingly smaller, more available, and more heterogeneous. Smart phones, tablets, smart watches, and a plethora of other such devices are being introduced and accepted by the consumer market at an accelerating pace. Computing has no value in and of itself, rather it relies on useful applications. When computing devices become more heterogeneous, then there is a problem in producing an incentive for software developers to produce new and port existing applications to them.

One of the main strategies employed in overcoming the problems introduced with machine diversity, is the specification of abstract, or conceptual, computing machines. The physical machines produced may then be designed to conform to one or more of these abstract machine specifications. This enables software developers to target these conceptual machines when developing applications, making them widely deployable on heterogeneous physical devices.

One such abstract machine is the WebGL [1] machine. WebGL is a promise specifying certain things a complying machine has to be able to do, at the issuing of specified instructions. WebGL was produced in order for developers to be able to use the Internet and web browsers as the primary means of distributing real-time graphical applications.

As of the day of the writing of this thesis, the WebGL 1.0 specification is conformed to, fully or in part, by many of the commonly used web browsers. The amount of applications relying on the standard is still, however, relatively small. The standard is expected to mature over time, and receive an increase of users and utility.

There is a great need to make the WebGL implementations more stable, and knowledge about how to effectively utilize them, more available; which is the primary incentive for the pursuing of this degree project.

3. N-Buffering

This section contains a brief theory on the need and utility of various configurations of data buffers in generals, and screen buffers in particular. In case the topics are familiar to the reader, the reader is invited to skip to the next chapter.

3.1 Buffering

Buffering is the act of queuing data between two processes. A producer process adds new data to the buffer memory region as data is completed, while a consumer process removes data from the memory region as it finishes processing previously removed data. Buffering is a decoupling strategy which make two processes, actual or conceptual, time-independent as long as the buffer memory region does not run out of space. The use of buffers is beneficial in avoiding processes having to wait for each other to enter a certain state in order to continue their operation, also usually referred to as synchronization.

Despite the primary purpose of buffers being to avoid synchronization, one of the main concerns while implementation them is avoiding the impact of the synchronization that is still required. The processes acting on the buffer must always be seeing the buffer in complete states, which means the buffer essentially is reserved to the one process currently using it. If both processes access the buffer frequently, then they will both have to wait to access it for significant amounts of time. This can be overcome by using several buffers, reserved in order by both processes.

As observed by M. Trapp in *OpenGL-Perfomance and Bottlenecks*, 2004 [4], the sending of data between two processes is the principal bottleneck structure of any processing pipeline. When some activity by either of two communicating processes requires significant time with exclusive buffer access, then there is significant chance that such activity will cause the other process to idle for a non-trivial amount of time.

Even though this chapter specifically investigates the buffer between a graphics machine and a monitor, the principles conveyed are widely applicable in many other processing pipeline contexts.

3.2 Screen Buffering

A modern computing machine and its attached screen, or monitor, are two conceptual processes acting on a single buffer, namely the frame buffer [5]. This frame buffer represents the link to the monitor attached to the computing device from the perspective of the computing device. By altering the contents of this buffer, the visible contents on the attached monitor are changed the next time the monitor is updated.

3.2.1 Single Buffer

If a computer application, as illustrated in Illustration 1, would perform its drawing operations directly to the frame buffer, with no regard to when the monitor is updated (i.e. without synchronization), there would be a significant risk that the monitor would update while some drawing operation was being carried out. The monitor would show partial draw operations. The computing machine would be, however, completely unhindered in its rendering.

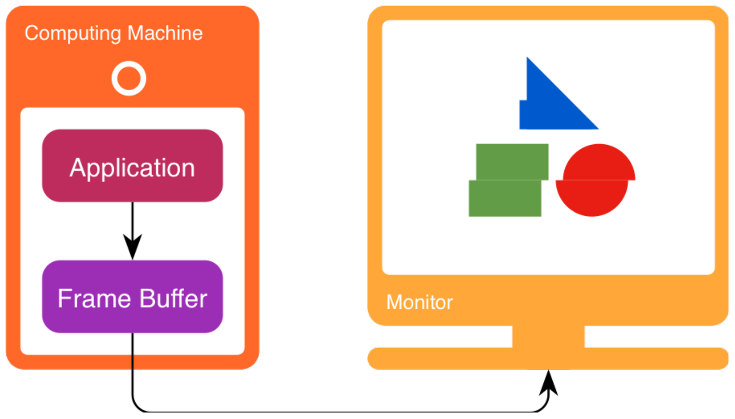


Illustration 1: Using a single buffer. May result in artifacts.

To avoid showing partial drawing operations to the device user, one strategy is to synchronize access to the frame buffer. This, however, leads to the computing application having to wait for significant amounts of time before being allowed to performed issued operations. Also, the computing application looses control over how many drawing operations are performed before the monitor is updated, which also may be regarded as the application showing partial results.

3.2.2 Double Buffers

To overcome the problems with single buffering, another buffer may be introduced, as depicted in Illustration 2. This second buffer is commonly referred to as a back buffer [5]. Rather than having the application draw directly to the frame buffer, it performs all of its operations to a back buffer, which has no direct relation to the monitor. When the application is done drawing to the back buffer, its contents are copied unto the frame buffer at a point in time when there is no risk of displaying incomplete results. The issue of exposing incomplete drawing operations is thus avoided.

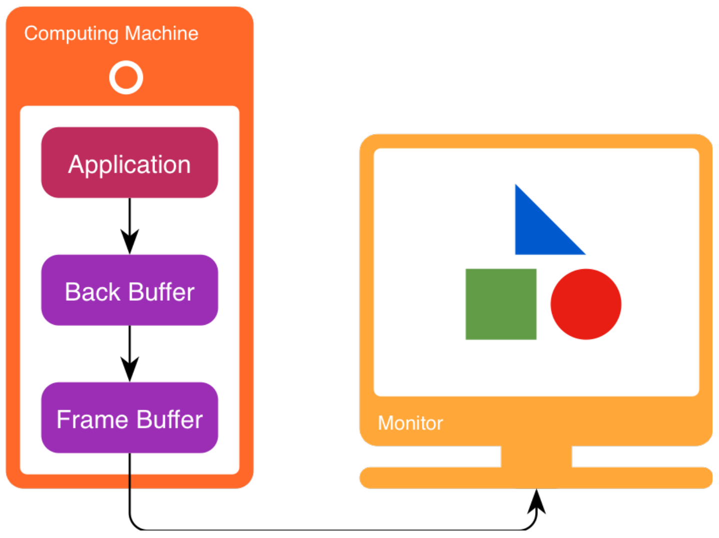


Illustration 2: Using double buffers. Avoiding artifacts.

3.2.3 Triple Buffers
Both using single or double buffers may, however, have some implications on performance. In both cases, access to the frame buffer may be synchronized.

By introducing a second back buffer [6], as illustrated in Illustration 3, a computer application can draw to that buffer immediately after completing operations on the first one, and thus avoid having to wait for the first one to become available again. The two buffers are used by the application in turn, and allows for the copying between a back buffer and the front buffer to happen to take some time.

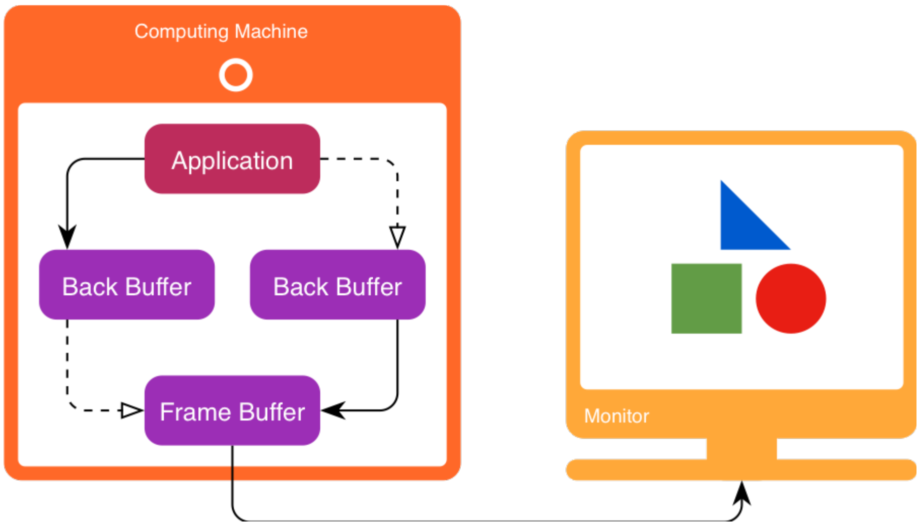


Illustration 3: Using triple buffers. Avoiding synchronization.

3.2.4 More Buffers
More buffers may be introduced to avoid rendering time jitter, which is fluctuations in rendering time. This, however, may lead to complications if rendering is to be done in immediate response to input, as the time between input being registered and its results are made visible is increased.

4. The Abstract WebGL Machine

This section briefly describes the environment and internals of the WebGL machine. In case the topics below are familiar they may be safely skipped, as the section contains no information regarding the aim of the project or its results.

The abstract WebGL machine was specified for the rendering of vector graphics using a dedicated hardware circuit, from an application residing within a web browser. The machine receives defined instructions in order to alter its behavior. In order to relate the standard in a brief manner, (1) the domains and languages of WebGL, (2) the machine internals, and (3) the machine application interface are hereafter described.

4.1 Application Stack

A WebGL application is, by nature, a distributed application. The application code and resources reside on a web server, and these are served as requested by eligible clients. This relies on code and resources being handled through a client application, a web browser, capable of correctly translating them into expected behavior. The WebGL specification [7] lists a set of technologies which a web browser has to conform to in order to properly run WebGL applications.

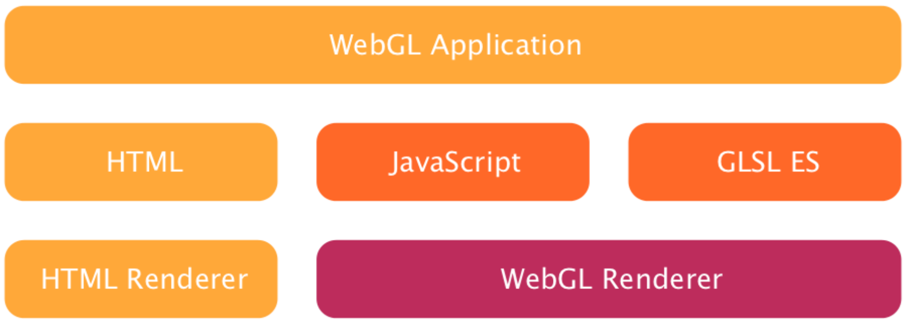


Illustration 4: The WebGL 1.0 Application Stack

The WebGL application stack consists of, as Illustrated in Illustration 4, the domain languages HTML, JavaScript and GLSL ES for WebGL. They define the logical context in which a WebGL application is run, and determine its behavior.

4.1.1 HTML

HTML [8], which is a language for specifying the logical layout and contents of a web page, provides the context in which a WebGL application resides. The HTML context has to contain an identified <*canvas*> element, from which a WebGL context may be acquired. The <*canvas*> element defines the size and position of the visible WebGL rendering context, and behaves as a regular HTML element.

4.1.2 JavaScript

JavaScript [9], which has been standardized under the name ECMA Script, is a simple prototype-oriented scripting language which may be used to manipulate the HTML Document Object Model (DOM) of a web page. The language is supported by most web browsers and is the only language currently available for use with the WebGL API, through which WebGL is utilized.

4.1.3 GLSL ES

GLSL ES [10], the OpenGL Shading Language for Embedded Systems, is the specification of two very similar languages. Code written in those languages is sent to, compiled by, and run by, the abstract WebGL machine. The languages define how the WebGL machine is to reply to issued instructions, and defines algorithms for calculating vertex transformations and color values. WebGL supports a subset of the OpenGL ES 2.0 Shading Language.

4.2 Rendering Pipeline

The rendering pipeline presented in this section explains the parts of the graphics machine which are imperative in understanding for the use of WebGL, and how these relates to incoming instructions and rendered content.

The abstract WebGL machine operates primarily on two kinds of data, which both share the same internal format. The first kind is geometry, or vertex data. A vertex is a relative point in space, which in WebGL has between two and four dimensions. The second kind of data, color, or fragment data, is described as sets of color values.

The rendering pipeline is the sequence of steps, transforming given geometry into pixel data, that are performed by the WebGL machine when it is issued a draw command. The sequence is depicted in Illustration 5.

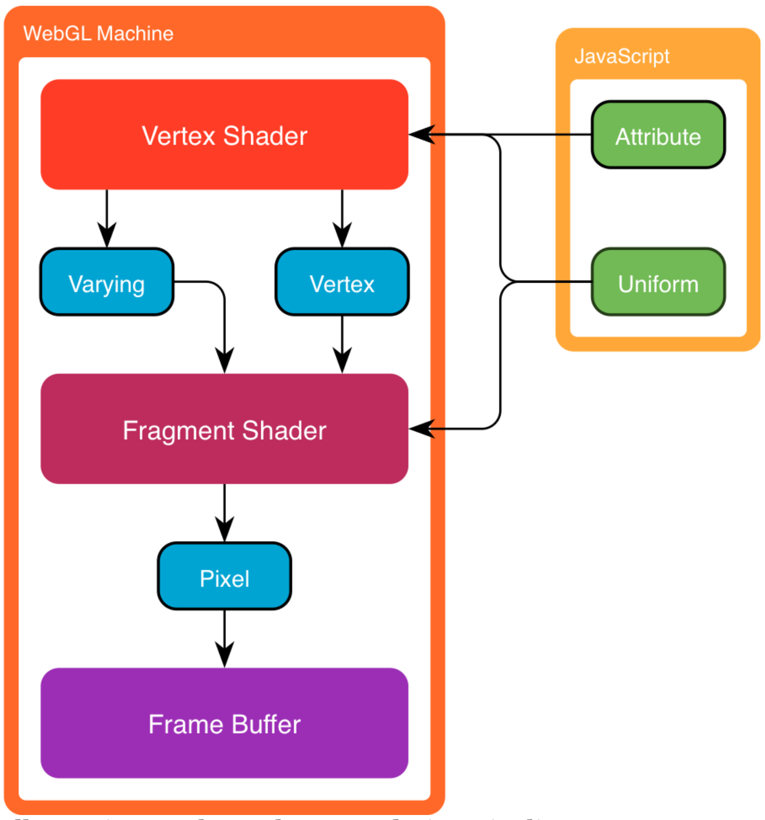


Illustration 5: The WebGL Rendering Pipeline.

Before a WebGL machine can operate on any vertex or color data, it has to have a shader program in place. The shader program consist of a vertex program and a fragment program, which are responsible for handling any data handed to the WebGL context. The above illustration depicts the relation between the shader programs, the frame buffer and the JavaScript context.

4.2.1 Setting Up State

Between each draw call, the state of the WebGL machine may be altered through the issuing of various calls through the JavaScript WebGL API. Apart from being able to upload texture data, manage buffers, and so on, room is given to pass values from the JavaScript context to the shader programs. These values are called **attributes** and **uniforms**. They differ through the former type being provided one value at a time, each value resulting in its own vertex shader call, and the latter type being provided to every single shader call, but not invoking calls on its own.

4.2.2 The Vertex Shader

All geometry provided, either directly or using buffers, travel one vertex at a time through the vertex shader. The primary purpose of the vertex shader is to determine which vertex value to pass on to the fragment shader. Vertices usually need to be adjusted in relation to the direction and perspective of the camera, and the state (position, rotation, etc.) of the model which the vertex belongs to.

The vertex shader also has the ability to pass on **varying** values directly to the fragment shader. These contain data such as colors, vector normals, or any other data which needs to be interpolated.

4.2.3 The Fragment Shader

The purpose of the fragment shader is to translate given geometry into fragment, or pixel data. In order to facilitate this, all data passed from the vertex shader to the fragment shader is interpolated. Interpolation is the process whereby all relevant intermediary values between two or more related values are calculated.

If, for example, the WebGL machine is instructed to draw a triangle, consisting of three vertices, these three vertices would result in three calls to the vertex shader. Subsequently, the fragment shader would have to be called the same amount of times as there are pixels between the vertices. The vertex passed on to each call of the fragment shader is an interpolated vertex from somewhere in-between the geometry vertices which make up the triangle.

When a suitable color is decided upon, in relation to, perhaps, light sources, materials, etc, it is passed on to the frame buffer.

4.2.4 The Frame Buffer

The frame buffer is the piece of memory which represent the *WebGLRenderingContext* in the hosting browser. When a pixel reaches the frame buffer, it is made visible at the next frame update.

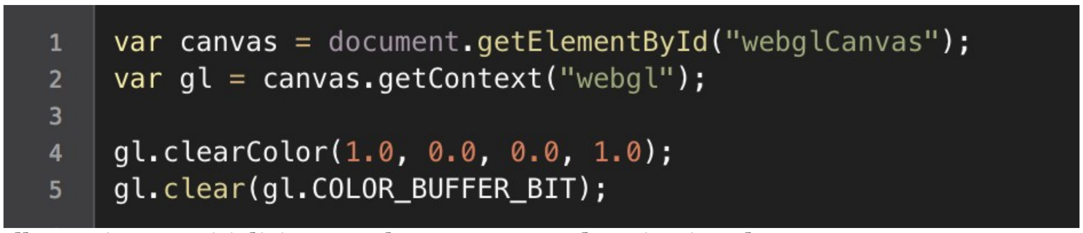
4.3 WebGL Application Interface (API)

The purpose of the JavaScript WebGL API [7] is to (1) initialize the WebGL context, (2) provide the context with data, and (3) to issue state calls to the context. The context is terminated automatically when no longer used.

4.3.1 Initialization

At the center of the WebGL API is the *HTMLCanvasElement* [11]. This element has to exist in the HTML file which include the JavaScript containing the WebGL business logic.

By calling the *getContext* method of the *<canvas>* element, the WebGL context is initialized, and an object containing the API methods is acquired.



```
var canvas = document.getElementById("webglCanvas");
var gl = canvas.getContext("webgl");

gl.clearColor(1.0, 0.0, 0.0, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT);
```

Illustration 6: Initializing a WebGL context and setting it red.

The above code example, Illustration 6, shows how to acquire a reference to a *HTMLCanvas-Element* object identified by *"webglCanvas"*, and then initialize a *WebGLRenderingContext*, which interfaces the WebGL API. For the sake of providing some visual feedback, the example also contains code for setting the color of the WebGL context to red.

4.3.2 Context Data

There are three main categories of data which may be communicated to the WebGL machine, (1) shader programs, (2) arbitrary number data, or primitive data, and (3) texture data.

Shader programs. In order for a WebGL machine to do anything more than change the color of a context, it must utilize a shader program. The GLSL ES code constituting the shader program is uploaded, compiled, and linked into a program residing inside the WebGL machine by a series of calls.

Primitive data. The WebGL machine will, in most common scenarios, require some data to work on in producing visual output. The WebGL API provides methods for uploading floating point numbers, integers, arrays, vectors, and matrices. Arrays of vectors are traditionally used for sending complex geometry to the WebGL machine, and matrices are often used for manipulating geometry models, the camera view and the camera projection. The WebGL standard allows, however, for any data to be used in any way. Data may be uploaded for use during one draw operation, or into buffers, allowing the data to be used without being sent until the buffer is destroyed.

Texture data. Textures are images mapped unto the space between vectors. Sending texture data to the WebGL machine relies on using *HTMLImageElement* objects. They are uploaded into buffers, and used by binding them before issuing draw calls.

4.3.3 Other State Calls

In one way or another, all methods of the WebGL API issue some command to the WebGL machine, even the ones used for initialization and context data uploading. The rest of them are used to set the abstract machine in different states. For example, calling *drawArrays()* will make the machine start rendering until previously issued operations are completed, calling *bindTexture*() makes the selected texture buffer available during the next rendering, and so on.

There is one implicit state call, which is worthy of mention. When the JavaScript code returns processing time to the browser, the browser will cause the <*canvas*> element hosting the WebGL context to be updated [12]. This means that swapping the frame and back buffers, or some other equivalent operation causing changes to be made visible, is performed implicitly by the HTML renderer and not explicitly from the JavaScript context.

5. Research on Synchronization

This chapter describes the research made on the subject of finding potential issues which may be countered using N-buffering. It depicts a naïve concrete rendering machine, and accounts for its complexities within the scope of the thesis. It also presents viable strategies in countering those complexities.

In case there are no points of synchronization in the abstract WebGL machine causing it to wait for significant amounts of instruction cycles, then there are no viable advantages in using triple buffers in relation to rendering performance, as explained in chapter 3. For this reason it is imperative to understand when and how synchronization occurs.

The primary source of information on this particular subject is the formal WebGL 1.02 specification. Secondarily, information published by OpenGL ES compliant graphics hardare vendors, such as Apple Inc, Blackberry Inc, NVIDIA, etc. may be used to gain understanding on the subject of how they view their own implementations as concrete machines, and which recommendations they give regarding programming such.

5.1 The Formal WebGL Machine

The abstract WebGL machine, as described in version 1.0.2 of the WebGL specification [13], is not defined independently, but rather as a subset of OpenGL ES 2.0.25 [14]. WebGL builds on the definitions in OpenGL ES, and then explicitly states the differences.

Searching all chapters and sections in both the WebGL 1.0.2 [13] and the OpenGL ES 2.0.25 [14] specifications for keywords such as “synchronize”, “flush”, “queue”, etc. the conclusion ought to be reached that the formal definition of a WebGL machine does not state *how* to construct a compliant machine. Rather, the formal specification is primarily concerned with *what* such a machine ought to be able to *do*. No specific details are outlined about command queues, pipelines, or any other concrete construct.

The WebGL specification does, however, define the functions *flush* and *finish*. The former forces the application to wait until all issued commands have been successfully communicated to the WebGL machine, and the latter forces waiting until the WebGL machine has completed any pending command. This implies that most WebGL implementations are expected to use a command queue, and that the graphics machine is expected to operate asynchronously in relation to the utilizing application.

5.2 Machine Vendor Recommendations

As WebGL is a standard complied to by a web browser, and not directly by a graphics card, some assumptions had to be made in relation to which graphics cards are relevant to research. Since WebGL is derived from OpenGL ES 2, the the scope of the study was limited to the category of graphics cards which comply with this standard. In reality, however, WebGL may be implemented on top of Direct X, OpenGL, or any other standard, with or without any relation to a dedicated hardware circuit.

5.2.1 OpenGL ES on Apple iOS

Apple Inc. has published guidelines for people developing applications utilizing OpenGL ES on their iOS devices [15]. The guidelines describe the graphics card as a server receiving commands from the application, taking the role of a client. Both parts are described as operating independently and communicating through client requests.

The guidelines also contained details regarding shared resource contention. In case information is altered by the client or read by the server at the same time, locking occurs. This is accordingly true for textures, vertex buffer objects (buffered primitive data), or any other buffered data. Also, N-buffering of data on the graphics card is suggested as an approach in avoiding synchronization.

5.2.2 OpenGL ES on BlackBerry 10

The technical guidelines for developing BlackBerry 10 OpenGL ES applications [16] are similar to those of Apple Inc. The guidelines concur on the image of the OpenGL ES machine as asynchronous and having command queues. The guidelines recommend striving for few draw operations per frame rendered, using double or more vertex buffer objects to avoid resource contention, and performing minimal amounts of synchronizing operations, such as *flush* or *finish*.

5.2.3 OpenGL ES on NVIDIA Tegra

Similar guidelines as those previously mentioned are available from the graphics chip manufacturer and designer NVIDIA Corporation. In the guidelines for their Tegra chips [17], the idea of a command queue is reaffirmed. The advice to "[t]ry to deliver as much geometry as possible with each submitted draw call" in the mentioned guidelines implies that there is a performance cost attached to performing such a draw operation.

5.3 A Notional Concrete Rendering Pipeline

As there is no reference implementation of WebGL, and a compliant machine may behave in any way necessary to fulfill the requirements of the specification, there is no definitive way of describing when synchronization will occur in the abstract, or imaginary, WebGL machine. As this is the case, a notional concrete rendering pipeline was produced from the observations and recommendations available in from the sources mentioned in sections 5.1 and 5.2. While it might be regarded as speculative, it is substantial enough to draw some theoretical conclusions about avoiding synchronization.

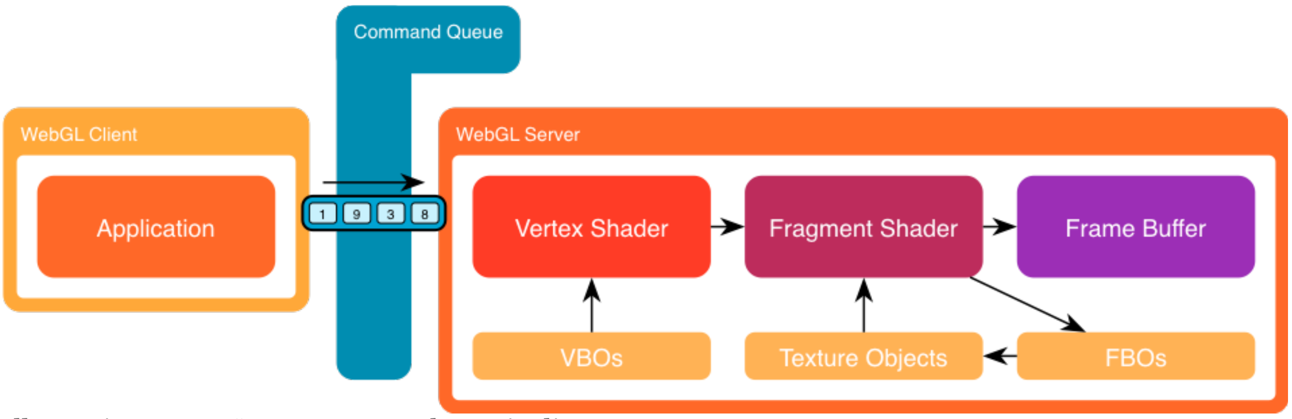


Illustration 7: A naïve concrete WebGL pipeline.

The notional machine, as illustrated in Illustration 7, has some significant characteristics in relation to synchronization, as outlined in the below subsections.

5.3.1 The Command Queue

Whenever the WebGL client application issues any command to the WebGL server, that command is put on the *Command Queue*. This queue exists in order for the client and server to be able to operate in parallel.

One way commands. Most commands available for putting on the command queue are one-way commands, and do not require the server to send any reply to the client.

Blocking commands. Some commands, such as *readPixels*, *getError*, *getParameter*, etc. are specified as immediately returning some data to the commanding client. Such commands will most likely cause the client application to idle while waiting for the server to finish all commands up until the one waited for. While waiting, the client application may not issue any more commands. This causes the server to be entirely out of work when the blocking command has been carried out.

Queue overflow. The command queue may be assumed to be finite in length. If an issued client command does not fit in the WebGL server, then the command has to be stored somewhere else while waiting for room in the command queue. The WebGL specification does not define any behavior in relation to queues, which means the WebGL implementation must guarantee that all issued commands are executed at some point. Synchronizing a client-side queue with the command queue may be costly and would be expected to have some impact on performance.

Changing state may be expensive. Each processed command implies a server state change. State changes may require tear-downs and initializations, reducing the effective time calculations may be performed.

5.3.2 Internal Buffers

The WebGL server houses its own buffers containing data uploaded by the client. The relevant kinds of buffers are VBOs (Vertex Buffer Objects), texture buffers, and FBOs (Frame Buffer Objects).

As a general rule, when the WebGL server and the WebGL client attempt to read from and write to the same buffer at the same time, synchronization occurs. The client will be forced to wait until the server is done reading. This particular kind of problem is called *resource contention*.

Resource contention is most probable in vertex buffer objects. Textures are more commonly uploaded during initialization, and freed during termination, and are only read by the graphics machine during runtime. Vertex buffers objects, however, contain geometry and other primitives, and will likely need to be altered very frequently in order to produce animation.

Frame buffer objects are, in essence, back buffers, as explained in chapter 3. The WebGL machine may be directed to draw unto such a buffer rather than drawing to the frame buffer. These buffers may be used for frame post-processing, or for generating textures mapped unto geometry. FBOs are unlikely to be subject to contention since they are both generated and consumed by the server.

5.4 Avoiding Synchronization Issues

With the ideas presented in section 5.3 in mind, which are the strategies most suitably deployed in order to compensate for the issues which may arise from synchronization?

5.4.1 Using N Frame Buffer Objects

Frame buffer objects may be utilized to render one or more frames before they are to be presented through the default frame buffer. This way the WebGL application is given time to compensate for fluctuations in waiting time, as explained in section 3.2.3.

One drawback of this approach is that it would impose additional delay before the application may visually represent feedback to a potential application user, feeding the application with instructions through means such as mouse, keyboard, gamepad, or similar. It would also lead to using up more memory on the WebGL server, which may be non-trivial in serving serving data to monitors with arbitrarily high pixel count, or in WebGL servers where the amount of memory available is limited.

5.4.2 Using N Vertex Buffer Objects

Since there is a significant risk of resource contention in frequently altered vertex buffer objects, a viable strategy in avoiding that would be to use N, but most probably two, vertex buffer objects. Since writing to a buffer is assumed to be more costly than reading from a buffer, there seems to little to gain in keeping more than two such buffers. The buffers would be used in turn by both client and server.

The only apparent drawbacks would be the increase in application complexity and the increase of memory usage.

5.4.3 Minimizing Communication

Through making sure that the server acquires the fewest possible amount of instructions for it to be able to perform its designated tasks, the risk of buffer overflow is reduced, and the amount of state changes is decreased. This could, for example, be achieved through implementing more complex shader programs, by using sprite maps, or merging geometry always rendered together.

There technical drawbacks would involve a potential increase in application complexity.

6. Performance Analysis of WebGL Machines

This chapter relates how test cases were decided upon which, within the scope of the thesis, represent different areas of usage of a WebGL machine. It also describes the metrics which were chosen to measure WebGL machine utilization.

6.1 Normative Test Cases

This section outlines the different prototypes to be produced for testing the impact of introducing N buffers. The prototypes represent test cases which burden the WebGL application in different ways, which should provide different insights.

The test cases were chosen in relation to the strategies listed in section 5.4. They are to reflect normative usage of a WebGL machine in some relevant way. The strategy named Minimizing Communication is, however, not measured since it is not directly related to using buffer configurations.

6.1.1 Cheap Square Cloud

Particle effects such as clouds, sparks, or rain, can be achieved in diverse ways. This test will, however, require the WebGL client to provide the coordinates and colors of every single particle rendered. The client application will be responsible for re-positioning each particle every frame, causing high amount of communication between the client and server. This very unlikely reflects many actual implementation of particle effects, since much of the calculation burden could be moved unto the graphics machine, which is likely to be much more effective at performing such calculations.

This is expected to reduce the impact of using N frame buffer objects, but increase the impact of using N vertex buffer objects. This because of significant traffic between server and client, but the rendering time of each particle instruction being insignificant.

6.1.2 Expensive Sphere Cloud

Calculating light is a task which puts significant burden on the graphics card, but requires little information provided from the WebGL client.

A prototype is to be produced where light is radiated from the middle of the screen to N spheres, with each sphere moving in a circle around the screen. The client will provide coordinates and colors for the spheres.

This ought to reduce the impact of using N vertex buffer objects, but have an increase in performance when using N frame buffer objects. This is expected because the traffic between the server and client ought to be far from the maximum capacity, while the time required to render a single particle ought to be significant.

6.2 Relevant Metrics

In order to measure the difference which the introduction of N buffers make, data will be collected, as outlined in this section, running tests with all relevant configurations of buffers.

The time it takes for a WebGL application to render and present a frame consumed by the frame buffer ought to be an adequate indication regarding the general performance of the WebGL application.

There are, however, some implications surrounding this metric. The method advocated by W3C for issuing continuous rendering instructions to a JavaScript context, using the function *requestAnimationFrame*, is limited to an animation rate which corresponds to the vertical synchronization of the monitor in which the WebGL context is displayed [18]. For most computers this means that the rendering time for one frame is limited to a minimum of about 1/60 seconds, or about 16.7 ms, assuming most computer monitors has a refresh rate of 60 times per seconds.

There are other functions which may be used for the purpose of frame animation [19], but these give little to no guarantees about being accurate. The purpose for the existence of *requestAnimationFrame* is for the web browser to be able to prioritize those calls above other events which have less of an impact on the user experience. For this reason, *requestAnimationFrame* is used by the prototypes produced as part of the thesis.

In order to compensate for the limit in minimum rendering time, prototypes will be designed to first determine some relevant rendering burden, which then will be same both for rendering with and without buffers. This way the relative difference may be used to measure the impact of buffering.

The rendering time aimed for when determining the performance burden will be 1/20 seconds, or ***50 ms***.

It is, however, worthy of mention that as the relative difference in performance between buffer configurations will be of most interest, it does in reality not matter much whether or not the original burden placed the no-buffer configuration will result in an average frame rendering time near 50 ms.

Average Frame Rendering Time will be calculated by sampling the frame rendering time each rendered frame, summing them, and then dividing the sum with the amount of frames collected.

$$\mathbf{T = F_{time\ sum} / F_{amount}}$$

T = Average time, F = Frame.

The frame rendering time average will be presented together with information about **sample count**, **standard deviation**, and **confidence interval**.

7. Prototype Implementation

This section details how the prototype was designed in relation to buffer configurations and data collection.

7.1 Object Orientation in JavaScript

JavaScript was chosen the implementation language for the produced prototypes. JavaScript diverts from many more traditional object oriented languages in too many ways to account for as part of this thesis. Despite this, however, the code produces for the prototypes will be referred to as classes, objects, and using other traditional object oriented terms. For a more throughout description of the JavaScript language, please turn to the Mozilla Foundation [9], or some other authority on the subject.

7.2 Implementing N Buffers

As explained in sections 5.4.1 Using N Frame Buffer Objects and 5.4.2 Using N Vertex Buffer Objects, the primary goal of the prototypes produced was to measure the impact of using one or two vertex buffers, and using one or two frame buffers. For this reason, two classes of interest were produced, namely *FBOBuffer* and *VBOBuffer*.

Upon instantiation, both *FBOBuffer* and *VBOBuffer* allocate all the potentially necessary buffers they encapsulate. The actually utilized buffers may consequently be changed during the lifetimes of the objects through calling their *setN()* methods.

FBOBuffer. The *FBOBuffer* class, as seen in Illustration 8, relies on capturing draw calls. By putting draw operations between calls to the *startCapture()* and *stopCapture()* methods, the draw operations are performed to an off-screen texture, rather than to the default frame buffer. When a captured texture is needed for rendering, it may be acquired through the *getNextBuffer()* method. When only using one texture buffer, *getNextBuffer()* will return the last captured texture. When increasing the texture buffer amount, *getNextBuffer()* will return the buffer which was captured first of the currently unreturned buffers.

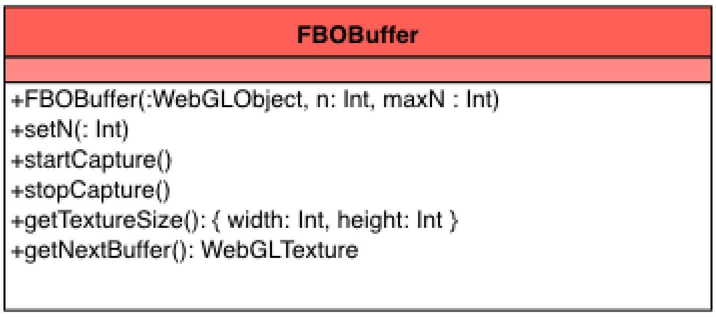

FBOBuffer
+FBOBuffer(:WebGLObject, n: Int, maxN : Int)
+setN(: Int)
+startCapture()
+stopCapture()
+getTextureSize(): { width: Int, height: Int }
+getNextBuffer(): WebGLTexture

Illustration 8: Frame Buffer Object Manager Class

VBOBuffer. Vertex buffers, which are encapsulated by the *VBOBuffer* class illustrated in Illustration 9, are used for storing data needed by a draw operation. For this reason the class in question contains the method *upload()*, which uploads given data into the vertex buffer used for the longest time ago. If only using one buffer, then that one is always used. Uploaded vertex data needs to be enabled before a draw call, and disabled after it. The methods *enable()* and *disable()* are used for activating and deactivating a relevant vertex buffer before and after a relevant draw operation.

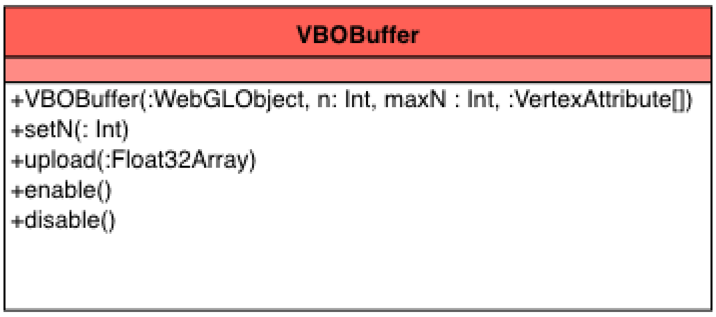

VBOBuffer
+VBOBuffer(:WebGLObject, n: Int, maxN : Int, :VertexAttribute[])
+setN(: Int)
+upload(:Float32Array)
+enable()
+disable()

Illustration 9: Vertex Buffer Object Manager Class

7.3 Collecting Data

Two categories of data was collected as part of the prototype benchmarks, average rendering times and relative performance indices. The data collected is motived in section 6.2 Relevant Metrics.

7.3.1 Average Rendering Times

Collecting the frame rendering times relied on the two classes in Illustration 10, *Analyzer* and *AnalyzerReport*. The first of those accepted rendering times through its *push()* method. After a benchmark was successfully completed, the *generateReport()* method was invoked in order to produce an instance of *AnalyzerReport*, which included all time samples, their sum, average and standard deviation.

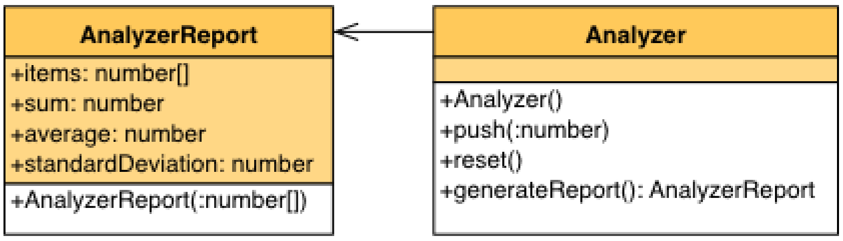


Illustration 10: Rendering Time Analysis Classes

All rendering times were collected in the main loop function, which was scheduled by the *requestAnimationFrame()* function, as described in section 6.2 Relevant Metrics.

7.3.2 Relative Performance Indices

A relative performance index is a burden, as a particle amount, placed upon the benchmark application in order to adjust the average frame rendering time of the normative benchmark run to some relevant interval. The performance index is of no direct value to the conclusion of the thesis, but is required to increase the quality of the frame rendering times sampled.

The motivation behind collecting relative performance indices if further discussed in section 6.2 Relevant Metrics.

7.4 Cloud Prototype

The source code of the Cloud Prototype may be reviewed using the information in Appendix 2.

At the heart of the Cloud Prototype is the *Benchmark* class, as shown in Illustration 11. The Benchmark class may configured with different amount of frame and vertex buffers through its methods *setFBOTextureN()* and *setVBON()*. A benchmark may be run in calibrating mode, which is used to determine a suitable particle burden for the static mode, which in turn is used to collect frame rendering times. Data collection is further explained in section 7.3 Collecting Data.

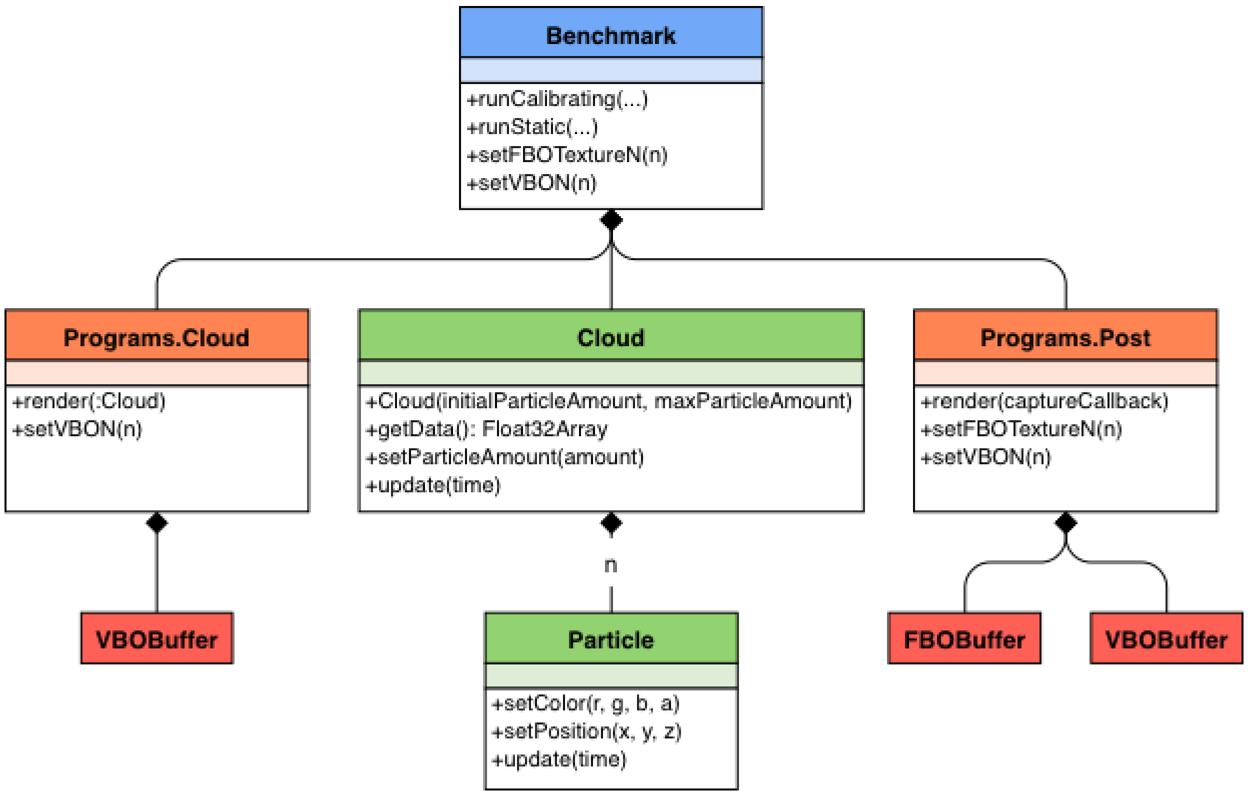


Illustration 11: Benchmark Class and Significant Encapsulated Classes

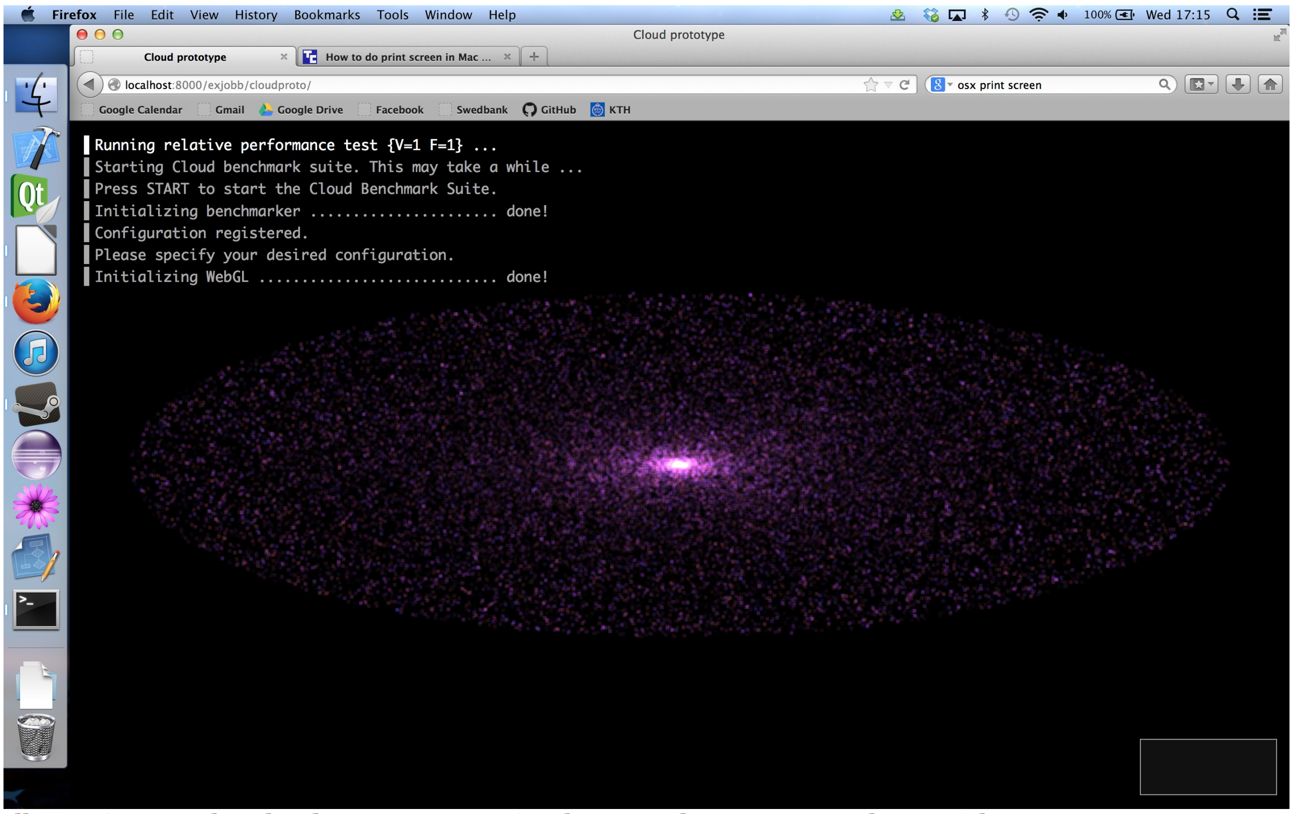
A *Benchmark* keeps track of two separate programs, a cloud program and a post program. The first program is used to render the particles in the cloud, while the second is used to manage the frame buffer. The cloud program keeps track of a *VBOBuffer*, which is used to upload particle information, while the post program keeps one *FBOBuffer* for keeping track of frame buffer textures, and one *VBOBuffer*, for keeping track of the four corners of the frame buffer textures.

The *Cloud* class manages a list of pre-initialized *Particle* objects. All particles store their data in a single *Float32Array*, which they all refer to internally. When calling the *getData()* method of the *Cloud* class, a view into the array containing all particle data is returned. This method is used by the *Programs.Cloud* class in order to present *Particle* data to the graphics machine. The positions of all *Particle* objects are updated by calling the *update()* method, which iterates through all active particles.

The *Benchmark* class may be initialized into two different modes, cheap and expensive mode. The only difference between these modes, is the complexity of the cloud shader program used. By using one program where the complexity of a particle is low, and one where the complexity of a particle is very high, the goals set out in section 6.1 were satisfied without the need to write two separate prototypes.

7.4.1 Cheap Mode

Running the Cloud Prototype in cheap mode, all particles are drawn as small transparent squares. An example of this is shown in Illustration 12.

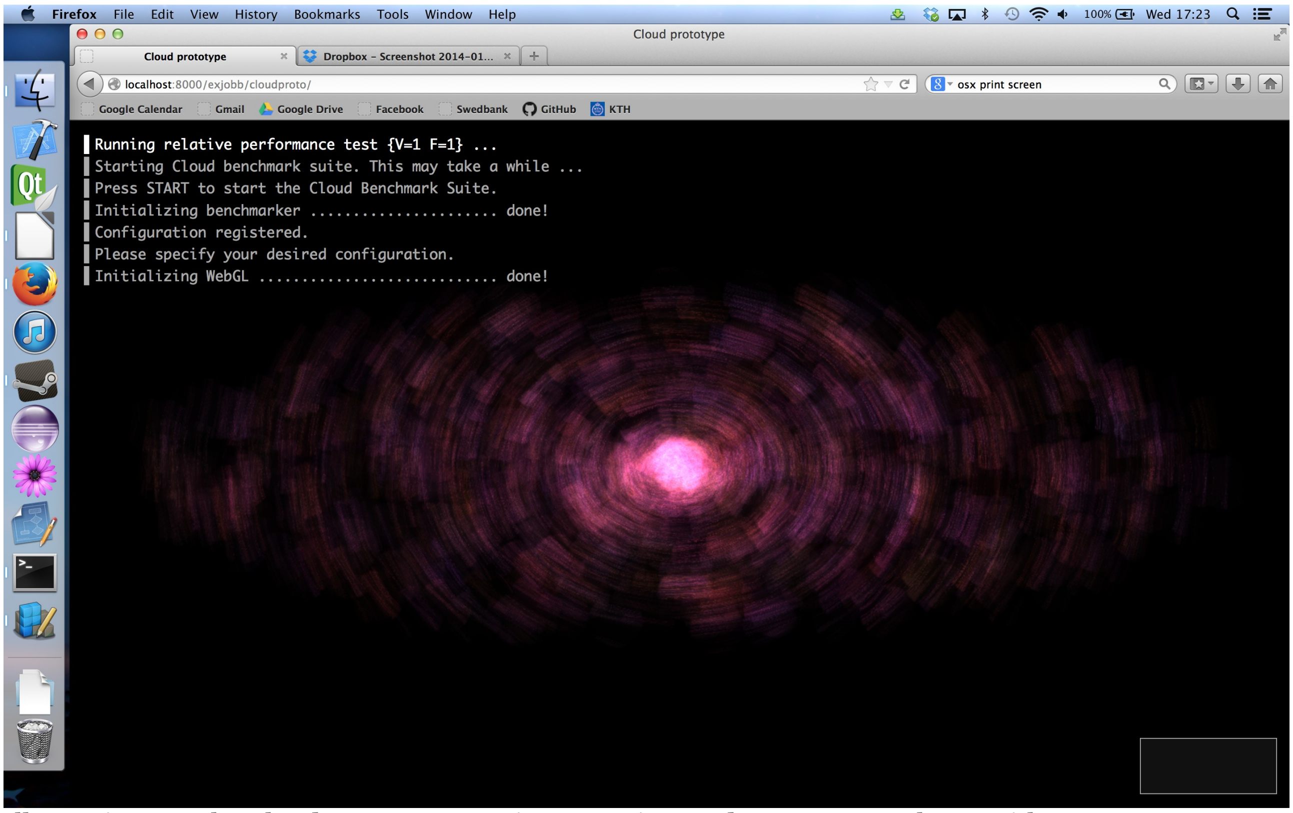


```
Running relative performance test {V=1 F=1} ...
Starting Cloud benchmark suite. This may take a while ...
Press START to start the Cloud Benchmark Suite.
Initializing benchmarker ..................... done!
Configuration registered.
Please specify your desired configuration.
Initializing WebGL ........................... done!
```

Illustration 12: The Cloud Prototype Run in Cheap Mode on a MacBook Pro with OS X 10.9.

7.4.2 Expensive Mode

When run in expensive mode, the particles of the Cloud Prototype are drawn as spheres drawn with a point light light source in the middle of the screen. Also, in order to further increase the stress on the graphics machine, a Mandelbrot pattern is drawn on top of each sphere. As exemplified in Illustration 13, most devices were unable to properly display the Mandelbrot pattern. The device used to capture Illustration 13 rather rendered the Mandelbrot pattern as complex noise.



```
Running relative performance test {V=1 F=1} ...
Starting Cloud benchmark suite. This may take a while ...
Press START to start the Cloud Benchmark Suite.
Initializing benchmarker ..................... done!
Configuration registered.
Please specify your desired configuration.
Initializing WebGL ........................... done!
```

Illustration 13: The Cloud Prototype Run in Expensive Mode on a MacBook Pro with OS X 10.9.

8. Data Results and Analysis

The circumstances in which data was gathered through the prototypes is related, some data of interest, as well as a brief analysis.

8.1 Environment

Web browsers. The two web browsers Mozilla Firefox and Google Chrome/Chromium were used for all tests. This primarily because both browsers were supported on all platforms the prototypes were performed on.

Devices. The prototypes were run on six different machines, whereof two were desktop computers, two were laptop computers, and two were portable smart devices. One laptop and one desktop computer ran the Ubuntu [20] operating system, versions 13.04 and 13.10, while the other two ran the Mac OS X [21] operating system, version 10.9. The two portable smart devices used, a tablet and a smart phone, were both running the Android [22] operating system, versions 4.3 and 4.4.

8.2 Notations

Buffers. The two different buffers described in section 5.4 Avoiding Synchronization Issues, are denoted, in the data presented below, with **V** and **F**, as in *Vertex buffer* and *Frame buffer* respectively. Each letter is followed by the amount of buffers used. **V1F2** would, for example, denote one vertex buffer and two frame buffers.

Time. All times are written in seconds. Average frame rendering times are shortened either to *Average Time* or *AVG.*

Confidence intervals. All charts have confidence intervals marked as T-shapes extending from the average values they consider. The degree of confidence for the intervals is 99.9%.

8.3 Cloud Prototype Results

The cloud prototype results of four different machine and browser combinations out of the total of twelve such are presented below. The remaining eight may be reviewed in Appendix 1.

Do note that the cheap and expensive prototypes results are separated and compared. As described in section 7.4 Cloud Prototype, the two prototypes burden the WebGL machine differently.

8.3.1 OSX10.9, MacBook Pro 2012

Firefox 27.0 (Cheap)

Buffer Setup	Samples	Average Time	Std. Deviation	Conf. Interval (99.9%)
V1 F1	1104	0.0543 s	0.0162 s	0.0543 ± 0.0016 s
V1 F2	1561	0.0384 s	0.0010 s	0.0384 ± 0.0001 s
V2 F1	1528	0.0393 s	0.0016 s	0.0393 ± 0.0001 s
V2 F2	1591	0.0377 s	0.0007 s	0.0377 ± 0.0001 s

Firefox 27.0 (Expensive)

Buffer Setup	Samples	Average Time	Std. Deviation	Conf. Interval (99.9%)
V1 F1	1131	0.0530 s	0.0062 s	0.0530 ± 0.0006 s
V1 F2	1131	0.0530 s	0.0065 s	0.0530 ± 0.0006 s
V2 F1	1130	0.0531 s	0.0064 s	0.0531 ± 0.0006 s
V2 F2	1130	0.0531 s	0.0078 s	0.0531 ± 0.0008 s

Table 1: OSX 10.9, MacBook Pro 2012 - Cloud Prototype Results

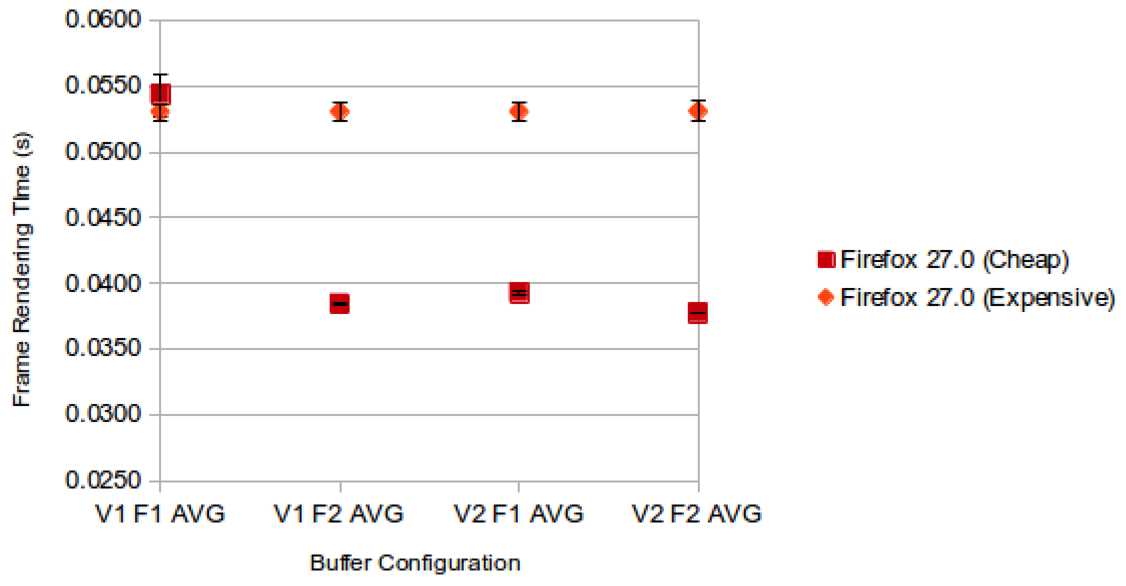


Chart 1: OSX 10.9, MacBook Pro 2012 - Cloud Prototype Results

As clearly seen in Table 1 and Chart 1, there is a significant performance gain on this particular machine in using any of the double buffer configurations, but only for the cheap prototype. Having both buffers doubled yields the best performance in this case, but only marginally better than having any of the buffers in a double configuration.

8.3.2 Ubuntu 13.04, Samsung X120

Chromium 31.0 (Cheap)

Buffer Setup	**Samples**	**Average Time**	**Std. Deviation**	**Conf. Interval (99.9%)**
V1 F1	1207	0.0497 s	0.0065 s	0.0497 ± 0.0006 s
V1 F2	1547	0.0388 s	0.0136 s	0.0388 ± 0.0011 s
V2 F1	1217	0.0493 s	0.0061 s	0.0493 ± 0.0006 s
V2 F2	1666	0.0360 s	0.0138 s	0.0360 ± 0.0011 s

Chromium 31.0 (Expensive)

Buffer Setup	**Samples**	**Average Time**	**Std. Deviation**	**Conf. Interval (99.9%)**
V1 F1	1240	0.0484 s	0.0102 s	0.0484 ± 0.001 s
V1 F2	1240	0.0484 s	0.0100 s	0.0484 ± 0.0009 s
V2 F1	1241	0.0483 s	0.0100 s	0.0483 ± 0.0009 s
V2 F2	1241	0.0483 s	0.0101 s	0.0483 ± 0.0009 s

Table 2: Ubuntu 13.04, Samsung X120 – Cloud Prototype Results

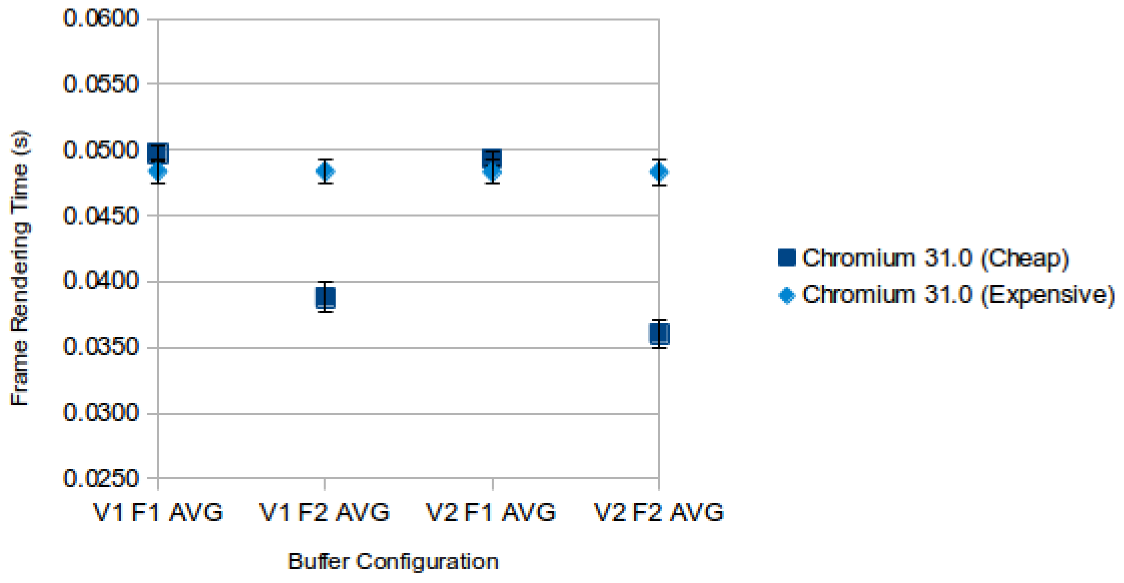


Chart 2: Ubuntu 13.04, Samsung X120 – Cloud Prototype Results

This machine yielded improvements similar to the MacBook Pro 2012 machine, with the difference that no substantial performance improvements could be made with only the vertex buffer being double. Having both buffers double yielded, however, better performance than just having the frame buffer double.

8.3.3 Ubuntu 13.10, Intel Core i5-4670T

Firefox 27.0 (Cheap)

Buffer Setup	**Samples**	**Average Time**	**Std. Deviation**	**Conf. Interval (99.9%)**
V1 F1	1133	0.0529 s	0.0043 s	0.0529 ± 0.0004 s
V1 F2	1114	0.0538 s	0.0049 s	0.0538 ± 0.0005 s
V2 F1	1128	0.0532 s	0.0055 s	0.0532 ± 0.0005 s
V2 F2	1145	0.0524 s	0.0053 s	0.0542 ± 0.0005 s

Firefox 27.0 (Expensive)

Buffer Setup	**Samples**	**Average Time**	**Std. Deviation**	**Conf. Interval (99.9%)**
V1 F1	1394	0.0430 s	0.0034 s	0.0430 ± 0.0003 s
V1 F2	1396	0.0430 s	0.0070 s	0.0430 ± 0.0006 s
V2 F1	1448	0.0414 s	0.0022 s	0.0414 ± 0.0002 s
V2 F2	1520	0.0395 s	0.0033 s	0.0395 ± 0.0003 s

Table 3: Ubuntu 13.10, Intel Core i5-4670T - Cloud Prototype Results

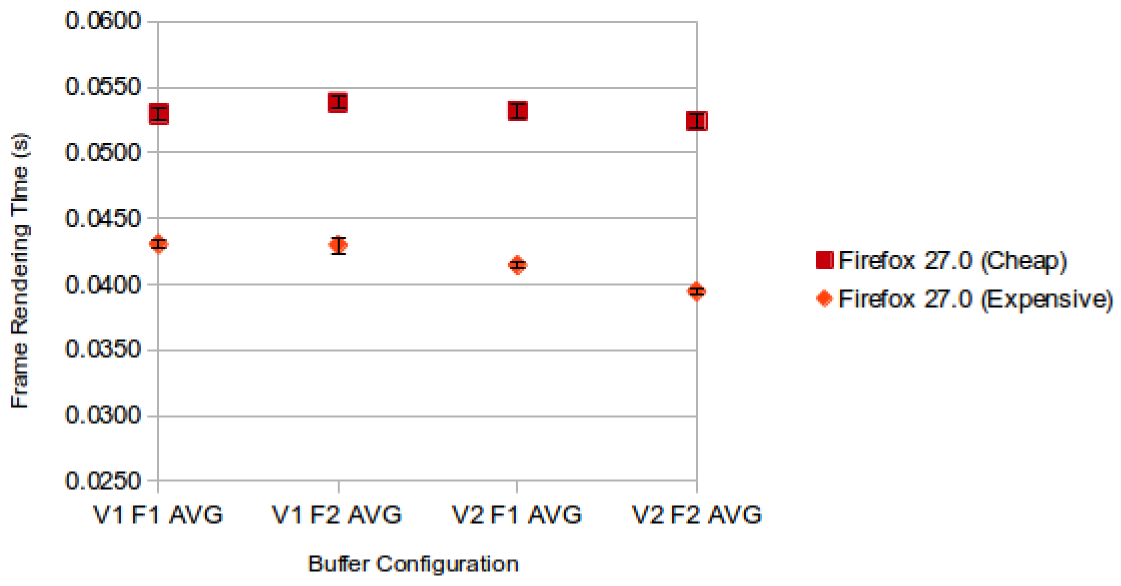


Chart 3: Ubuntu 13.10, Intel Core i5-4670T - Cloud Prototype Results

In contrast to the other results, this machine did not yield better performance when using double buffers in the case of the cheap prototype results. No buffer configuration had any major significant impact. Another difference between this and the earlier results is that the expensive prototype yielded better performance, for two buffer configurations. This especially in the case of having both buffers double, but also when having just double vertex buffers.

8.3.4 Android 4.4, Nexus 4 (2012)

Chrome 33.0 (Cheap)

Buffer Setup	**Samples**	**Average Time**	**Std. Deviation**	**Conf. Interval (99.9%)**
V1 F1	1214	0.0494 s	0.0237 s	0.0494 ± 0.0022 s
V1 F2	1420	0.0423 s	0.0200 s	0.0423 ± 0.0017 s
V2 F1	1405	0.0427 s	0.0199 s	0.0427 ± 0.0017 s
V2 F2	1434	0.0419 s	0.0193 s	0.0419 ± 0.0017 s

Chrome 33.0 (Expensive)

Buffer Setup	**Samples**	**Average Time**	**Std. Deviation**	**Conf. Interval (99.9%)**
V1 F1	372	0.1611 s	0.0734 s	0.1611 ± 0.0125 s
V1 F2	371	0.1615 s	0.0654 s	0.1615 ± 0.0112 s
V2 F1	371	0.1615 s	0.0701 s	0.1615 ± 0.0112 s
V2 F2	370	0.1620 s	0.0750 s	0.1620 ± 0.0128 s

Table 4: Android 4.4, Nexus 4 (2012) - Cloud Benchmark Results

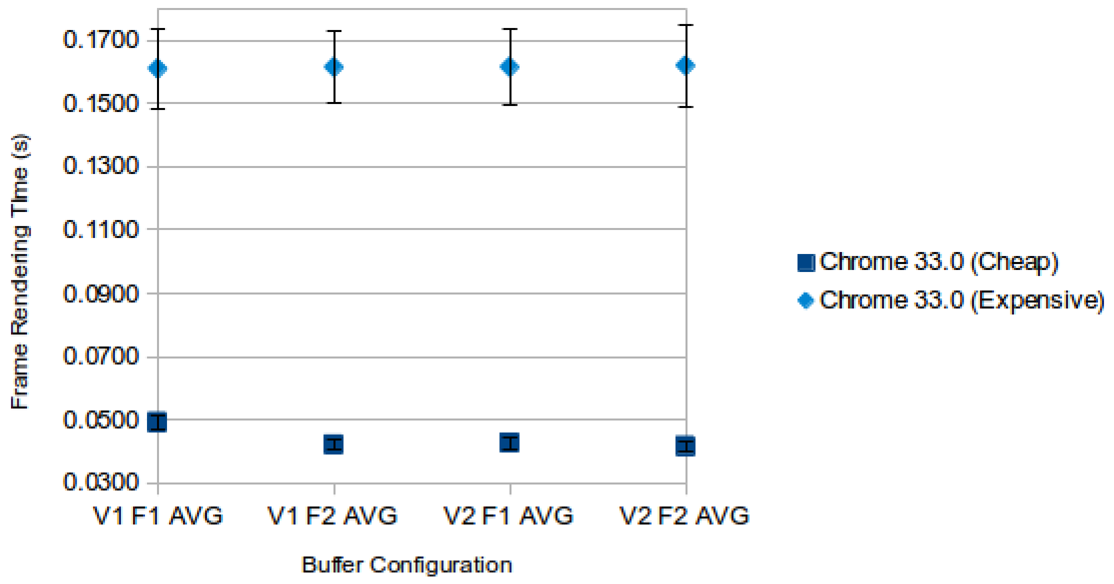


Chart 4: Android 4.4, Nexus 4 (2012) - Cloud Benchmark Results

The Nexus 4 (2012) smart device yielded results similar to the MacBook Pro 2012 machine. Worthy of special mention is that the device was barely able to run the expensive prototype, which is the reason for those the average rendering times having quite the wide confidence interval, and also being significantly higher than the target 0.05 s average aimed at for the first configuration.

8.4 Analysis

8.4.1 Stability Issues

Externalities. It was noted while running the different prototypes on the different devices that screen savers, notifications, and other externalities could impact the results while the benchmarks were being performed. Even though precautions were taken against such, there may have been significant disturbance from such unknown to the author.

GLSL ES compliance. Also, it was only one device which was able to render the expensive cloud prototype without significant artifacts, making the Mandelbrot pattern visible on the rendered spheres, namely the Nexus 4 device, running the Android 4.4 operating system, using the Chrome browser. Despite this being the case, all graphics cards ought to have executed the issued instructions, producing an equivalent burden in all cases. This cannot, however, be assured by comparing the visuals displayed while running the prototype.

8.4.2 Machine Diversity and Performance Gains

It was noted that even though several of the tested machines yielded results reminiscent of each other, there were some exceptions. The observation that WebGL graphics machine implementations differ in which optimizations from which they benefit was important in relation to the conclusion of the thesis.

9. Discussion

9.1 Implications of Increasing WebGL Performance

Increasing the performance of WebGL machines implies that machines of such application may render graphics either (1) at an increased frame rate, (2) with increased geometric detail, or (3) using less processing time. This may lead to less potent hardware performing more sophisticated rendering, and hardware consuming less power. These characteristics of improved performance lead to improved utility and reduced costs, as battery times are prolonged and cheaper circuits may be utilized. These are important economic and environmental incentives for performing such optimization.

9.2 Further Increasing WebGL Performance

As of the date of the writing this thesis, there is a significant performance and stability gap between the WebGL standard and the more traditional OpenGL and OpenGL ES standards. Investigating viable optimization strategies is one part in minimizing this gap, but the other would be to try to replace inefficient parts of the WebGL application stack.

As suspected by the author of the thesis, the JavaScript implementation would be the most significant inefficiency of the WebGL stack, and would either have to be made more efficient, or replaced.

Pursuing this particular ambition, the Mozilla Foundation and Google, producing the web browsers Firefox [2] and Chrome/Chromium [23], have invested significant effort in making their JavaScript interpreters more efficient.

Additionally, Google has launched the projects NaCl (Native Client) [24], which allows running C/C++ through a sandboxed environment in their browser, and Dart [25], which is essentially is a byte-code format intended as an alternative to JavaScript. Both projects may potentially replace JavaScript and allow access to an API for communicating directly with graphics hardware. Both projects are, however, early in their development and are not supported by other web browsers.

Mozilla Foundation has been investing effort in their ASM.js project [26], which is a JavaScript superset which allows JavaScript to be used as a byte-code format. ASM.js allows developers to write C/C++ code, or some other supported language, and compile it into ASM.js.

Independent of the successes of the mentioned projects, it is likely that JavaScript either will be replaced or made more efficient in order to increase the performance of the standard WebGL application stack during the years to come.

9.3 Recommendations for Further Research

If indeed, as stated in Section 9.2, JavaScript being slow is the reason for the buffer configurations not working as intended, then speeding up JavaScript considerably ought to increase N buffer impact. This could be done by producing prototypes similar to the ones produced as part of this thesis and compiling them to ASM.js [26] from C/C++ code. Also, Google NaCl [24] or Dart [25] could also be relevant to subject to similar research.

Also, to acquire a better understanding of the optimizations already made by the web browsers internally, contact ought to be made through the open source communities/corporations producing them. Such contact could also yield information advantageous in ways unknown to the author of this thesis.

10. Conclusion

The conclusions made, by the author of the thesis, in relation to the usefulness and implications of using N buffers in relation to WebGL.

10.1 Optimizing WebGL Performance

The goal and purpose of the thesis was to answer whether or not N buffers could be used in increasing rendering performance of an abstract WebGL machine. The buffer configurations investigated were using double vertex buffer objects, and using double frame buffer objects.

The answer to this question would be that the strategies employed do improve performance, but not consistently across heterogeneous devices.

As it was observed that due to machine implementation differences, and the difference in burden placed on a particular machine by different applications, the gains of optimizations using N buffers varies quite a lot. It rarely seemed the case, however, that N buffer configurations yielded worse performance.

In contrast to the theory of the author, the cases where the traffic through the client-server pipeline were higher, benefited more from any of the two buffer configurations being double. The cases were the major part of the burden were on the WebGL machine itself seemed to gain very little from having N>1 buffers, if at all.

References

1. **About WebGL** [web], Khronos Group; 2013.
   [Last verified 2013-11-04]
   http://www.khronos.org/webgl/
2. **Firefox Features** [web], The Mozilla Foundation; 2013.
   [Last verified 2013-11-12]
   http://www.mozilla.org/en-US/firefox/features/
3. **Require.js – Home** [web], James Burke; 2014.
   [Last verified 2014-01-04]
   http://requirejs.org/
4. Mathias Trapp, *OpenGL-Performance and Bottlenecks*, **Chpt. 3. Bottlenecks**.
   Seminar paper, Hasso-Plattner-Institut at the University of Potsdam; 2004.
   [Last verified 2014-06-11]
   http://www.matthias-trapp.de/
5. **Double Buffering and Page Flipping** [web], Oracle; 2013.
   [Last verified 2013-11-05]
   http://docs.oracle.com/javase/tutorial/extra/fullscreen/doublebuf.html
6. **Triple Buffering** [web], Microsoft; 2013.
   [Last verified 2013-11-05]
   http://msdn.microsoft.com/en-us/library/windows/hardware/ff570099%28v=vs.85%29.aspx
7. **The WebGL 1.0 Specification, Editor's Draft** [web], Khronos Group; 2013.
   [Last verified 2013-11-12]
   http://www.khronos.org/registry/webgl/specs/latest/1.0/
8. **HTML & CSS** [web], The World Wide Web Consortium (W3C); 2013.
   [Last verified 2013-11-12]
   http://www.w3.org/standards/webdesign/htmlcss
9. **JavaScript** [web], The Mozilla Foundation; 2013.
   [Last verified 2013-11-12]
   https://developer.mozilla.org/en-US/docs/JavaScript
10. *GLSL ES 1.0.17 Specification*, **2 Overview of OpenGL ES Shading** [web/pdf],
    Khronos Group; 2013.
    [Last verified 2013-11-12]
    http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf
11. **HTMLCanvasElement** [web], The Mozilla Foundation; 2013.
    [Last verified 2013-11-12]
    https://developer.mozilla.org/en/docs/Web/API/HTMLCanvasElement
12. Kouichi Matsuda, Roger Lea, **No Need to Swap Buffers in WebGL**, appendix A.
    Addison Wesley, 2013, First edition, ISBN-13: 978-0-231-90292-4.
13. **The WebGL 1.0.2 Specification** [web], Khronos Group; 2013.
    [Last verified 2013-11-18]
    https://www.khronos.org/registry/webgl/specs/1.0.2/
14. **OpenGL ES 2.0.25 Specification**, [web/pdf], Khronos Group; 2013.
    [Last verified 2013-11-18]
    http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf
15. **OpenGL ES Design Guidelines**, [web], Apple Inc; 2013.
    [Last verified 2013-11-19]
    https://developer.apple.com/library/ios/documentation/3ddrawing/conceptual/opengles_
    programmingguide/OpenGLESApplicationDesign/OpenGLESApplicationDesign.html

16. **Best Practices: OpenGL ES**, [web], BlackBerry Limited; 2013.
[Last verified 2013-11-19]
http://developer.blackberry.com/native/documentation/core/best_practices_opengles.html

17. **Maximizing the GPU and CPU/GPU Parallelism**, [web], NVIDIA Corporation; 2013.
[Last verified 2013-11-19]
http://docs.nvidia.com/tegra/Content/GLES2_Perf_Maximize_GPU.html

18. **Timing control for script-based animations**, [web], The World Wide Web Consortium (W3C); 2013.
[Last verified 2013-11-21]
http://www.w3.org/TR/2013/CR-animation-timing-20131031/

19. **HTML5 Timers**, *working draft*, [web], The World Wide Web Consortium (W3C); 2013.
[Last verified 2013-11-22]
http://www.w3.org/TR/2011/WD-html5-20110525/timers.html#timers

20. **Ubuntu Desktop Overview**, [web], Canonical Ltd; 2013.
[Last verified 2013-12-17]
http://www.ubuntu.com/desktop

21. **Mac OS X - Technical Specifications**, [web], Apple Inc; 2013.
[Last verified 2013-12-17]
http://www.apple.com/osx/specs/

22. **About Android**, [web], Google Inc; 2013.
[Last verified 2013-12-17]
http://developer.android.com/about/index.html

23. **The Chromium Projects – Chromium**, [web], Google Inc; 2014.
[Last verified 2014-01-04]
http://www.chromium.org/Home

24. **Native Client – Technical Overview**, [web], Google Inc; 2014.
[Last verified 2014-01-04]
https://developers.google.com/native-client/dev/overview

25. *Dart: Up and Running*, **Chapter 1. Quick Start**, [web], O'Reilly/Google Press; 2014.
[Last verified 2014-01-04]
https://www.dartlang.org/docs/dart-up-and-running/contents/ch01.html

26. **ASM.JS Specification – Introduction**, [web], Mozilla Foundation; 2014.
[Last verified 2014-01-04]
http://asmjs.org/spec/latest/#introduction

Appendix

1. Benchmark Results

Frame Time Benchmarks – Cloud Prototype

CONFIDENCE ALPHA 0.0010 **V**n = Vertex buffer amount. **F**n = Frame buffer amount.

#	Profile	Browser/Program	Notes	Particles	V1 F1 AMT	V1 F2 AMT	V2 F1 AMT	V2 F2 AMT
1	Ubuntu 13.10, Intel Core i5-4670T	Firefox 27.0 (Cheap)		3568	1133	11140	1128	1145
2	Ubuntu 13.10, Intel Core i5-4670T	Firefox 27.0 (Expensive)	Artifacts.	603	1394	1396	1448	1520
3	Ubuntu 13.10, Intel Core i5-4670T	Chromium 32.0 (Cheap)		168343	1217	1353	1213	1439
4	Ubuntu 13.10, Intel Core i5-4670T	Chromium 32.0 (Expensive)	Artifacts.	16982	1158	1171	1155	1162
5	OSX10.9, MacBook Pro 2012	Firefox 27.0 (Cheap)		471072	1104	1561	1528	1591
6	OSX10.9, MacBook Pro 2012	Firefox 27.0 (Expensive)	Artifacts.	1814	1131	1131	1130	1130
7	OSX10.9, MacBook Pro 2012	Chrome 33.0 (Cheap)		333981	1278	1526	1521	1526
8	OSX10.9, MacBook Pro 2012	Chrome 33.0 (Expensive)	Artifacts.	1724	1171	1170	1171	1167
9	OSX10.8, MacMini 2011	Firefox 27.0 (Cheap)		378216	954	975	965	985
10	OSX10.8, MacMini 2011	Firefox 27.0 (Expensive)	Artifacts.	744	515	515	515	515
11	OSX10.8, MacMini 2011	Chrome 33.0 (Cheap)		182014	1258	1551	1549	1541
12	OSX10.8, MacMini 2011	Chrome 33.0 (Expensive)	Artifacts.	1045	1201	1223	1201	1796
13	Ubuntu 13.04, Samsung X120	Firefox 26.0 (Cheap)		12504	1194	1170	1185	1181
14	Ubuntu 13.04, Samsung X120	Firefox 26.0 (Expensive)	Black.	486	1020	1008	1012	1005
15	Ubuntu 13.04, Samsung X120	Chromium 31.0 (Cheap)		45784	1207	1547	1217	1666
16	Ubuntu 13.04, Samsung X120	Chromium 31.0 (Expensive)	Black.	891	1240	1240	1241	1241
17	Android 4.3, Nexus 7 (2012)	Firefox 25.0 (Cheap)		28705	1239	1284	1281	1267
18	Android 4.3, Nexus 7 (2012)	Firefox 25.0 (Expensive)	Failed.	0	0	0	0	0
19	Android 4.3, Nexus 7 (2012)	Chrome 30.0 (Cheap)		7969	1229	1229	1236	1227
20	Android 4.3, Nexus 7 (2012)	Chrome 30.0 (Expensive)	Failed.	0	0	0	0	0
21	Android 4.4, Nexus 4 (2012)	Firefox 27.0 (Cheap)		33412	1370	1387	1373	1403
22	Android 4.4, Nexus 4 (2012)	Firefox 27.0 (Expensive)	Failed.	0	0	0	0	0
23	Android 4.4, Nexus 4 (2012)	Chrome 33.0 (Cheap)		23300	1214	1420	1405	1434
24	Android 4.4, Nexus 4 (2012)	Chrome 33.0 (Expensive)	Artifacts.	936	372	371	371	370

#	V1 F1 AVG	V1 F2 AVG	V2 F1 AVG	V2 F2 AVG	V1 F1 STDEV	V1 F2 STDEV	V2 F1 STDEV	V2 F2 STDEV	V1 F1 CONF	V1 F2 CONF	V2 F1 CONF	V2 F2 CONF
1	0.0529	0.0538	0.0532	0.0524	0.0043	0.0049	0.0055	0.0053	0.0004	0.0002	0.0005	0.0005
2	0.0430	0.0430	0.0414	0.0395	0.0034	0.0070	0.0022	0.0033	0.0003	0.0006	0.0002	0.0003
3	0.0493	0.0443	0.0495	0.0417	0.0017	0.0040	0.0017	0.0014	0.0002	0.0004	0.0002	0.0001
4	0.0518	0.0512	0.0519	0.0516	0.0147	0.0149	0.0149	0.0150	0.0014	0.0014	0.0014	0.0014
5	0.0543	0.0384	0.0393	0.0377	0.0162	0.0010	0.0016	0.0007	0.0016	0.0001	0.0001	0.0001
6	0.0530	0.0530	0.0531	0.0531	0.0062	0.0065	0.0064	0.0078	0.0006	0.0006	0.0006	0.0008
7	0.0470	0.0393	0.0395	0.0393	0.0049	0.0042	0.0039	0.0039	0.0004	0.0004	0.0003	0.0003
8	0.0512	0.0512	0.0512	0.0514	0.0025	0.0055	0.0024	0.0024	0.0002	0.0005	0.0002	0.0002
9	0.0630	0.0616	0.0622	0.0609	0.0156	0.0161	0.0158	0.0159	0.0017	0.0017	0.0017	0.0017
10	0.1163	0.1164	0.1164	0.1164	0.0093	0.0102	0.0137	0.0102	0.0013	0.0015	0.0020	0.0015
11	0.0477	0.0387	0.0387	0.0390	0.0045	0.0041	0.0041	0.0042	0.0004	0.0003	0.0003	0.0003
12	0.0499	0.0490	0.0499	0.0334	0.0029	0.0048	0.0026	0.0053	0.0003	0.0005	0.0002	0.0004
13	0.0502	0.0513	0.0506	0.0508	0.0013	0.0061	0.0026	0.0014	0.0001	0.0006	0.0002	0.0001
14	0.0588	0.0595	0.0593	0.0597	0.0027	0.0030	0.0026	0.0031	0.0003	0.0003	0.0003	0.0003
15	0.0497	0.0388	0.0493	0.0360	0.0065	0.0136	0.0061	0.0138	0.0006	0.0011	0.0006	0.0011
16	0.0484	0.0484	0.0483	0.0483	0.0102	0.0100	0.0100	0.0101	0.0010	0.0009	0.0009	0.0009
17	0.0484	0.0467	0.0468	0.0474	0.0088	0.0030	0.0016	0.0051	0.0008	0.0003	0.0001	0.0005
18	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
19	0.0489	0.0488	0.0486	0.0489	0.0401	0.0402	0.0399	0.0402	0.0038	0.0038	0.0037	0.0038
20	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
21	0.0438	0.0433	0.0437	0.0428	0.0068	0.0059	0.0065	0.0048	0.0006	0.0005	0.0006	0.0004
22	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
23	0.0494	0.0423	0.0427	0.0419	0.0237	0.0200	0.0199	0.0193	0.0022	0.0017	0.0017	0.0017
24	0.1611	0.1615	0.1615	0.1620	0.0734	0.0654	0.0701	0.0750	0.0125	0.0112	0.0120	0.0128

2. Cloud Prototype Source Code Repository

The code may be browsed at the following internet URL, and may also be acquired using GIT with the same address: *https://github.com/emanuelpalm/cloudproto*.