|  |
| --- |
|  |
| An assessment of performance loss when WebGL is compared to OpenGL |
| James Morrissey Supervisor Robert Sheehy |
|  |
|  |
|  |

|  |
| --- |
|  |

# Abstract

Due to the advances in web based technologies, advanced interactive web applications are indistinguishable from their native desktop counterparts. OpenGL and WebGL are interactive graphics environments that normally run on a desktop environment and can now accessed via a web browser and any mobile devices.

Before WebGL, browsers required external proprietary plugins to be installed, in order view complex 3D graphics within the browser. Currently WebGL is integrated into most browsers. This paper will perform an in depth analysis on how WebGL accesses the graphics hardware on a computer via the web browser. Both OpenGL and WebGL will be evaluated to identify where they differentiate and compare the results. The architecture of both of these API’s will be evaluated, their CPU and GPU metrics will be measured.

Table of Contents

[Abstract 1](#_Toc410916263)

[Literature Review 6](#_Toc410916264)

[Chapter 1: Introduction 6](#_Toc410916265)

[Chapter 2: Computer Graphics 8](#_Toc410916266)

[Applications 9](#_Toc410916267)

[GPU 9](#_Toc410916268)

[CPU 9](#_Toc410916269)

[Designing efficient applications 9](#_Toc410916270)

[Modern Rendering pipeline 10](#_Toc410916271)

[Modern OpenGL 12](#_Toc410916272)

[GL Shading Language 12](#_Toc410916273)

[Control flow within GLSL 13](#_Toc410916274)

[Shaders 13](#_Toc410916275)

[Procedure to create a program 15](#_Toc410916276)

[Execution model of shaders 16](#_Toc410916277)

[Communication between shaders 16](#_Toc410916278)

[Geometry Stage 17](#_Toc410916279)

[Vertex Specification 17](#_Toc410916280)

[Vertex buffer objects (Buffer Objects) 18](#_Toc410916281)

[Vertex Array Object 18](#_Toc410916282)

[Vertex Processing 18](#_Toc410916283)

[Rasterization 19](#_Toc410916284)

[Viewport transform 20](#_Toc410916285)

[Back-face culling 20](#_Toc410916286)

[Frustum culling 21](#_Toc410916287)

[Per-Fragment Processing 21](#_Toc410916288)

[Pixel Ownership Test 21](#_Toc410916289)

[Scissor Test 22](#_Toc410916290)

[Alpha Test 22](#_Toc410916291)

[Stencil Test 22](#_Toc410916292)

[Depth Test 22](#_Toc410916293)

[Alpha blended 22](#_Toc410916294)

[Dithering 22](#_Toc410916295)

[Logical Operations 22](#_Toc410916296)

[Write Mask 23](#_Toc410916297)

[Output Merging 24](#_Toc410916298)

[Framebuffer 24](#_Toc410916299)

[Double buffering 24](#_Toc410916300)

[Depth buffer 25](#_Toc410916301)

[Stencil buffer 25](#_Toc410916302)

[Accumulation buffer 25](#_Toc410916303)

[Multisample buffer 26](#_Toc410916304)

[Auxiliary buffer 26](#_Toc410916305)

[Bottlenecks 26](#_Toc410916306)

[Chapter 3: OpenGL 27](#_Toc410916307)

[Introduction 27](#_Toc410916308)

[Background 27](#_Toc410916309)

[Fundamentals 28](#_Toc410916310)

[Coordinate system 28](#_Toc410916311)

[Normalized device coordinates 32](#_Toc410916312)

[Transformations and Matrices 32](#_Toc410916313)

[Open GL Architecture 34](#_Toc410916314)

[OpenGL architecture within a network environment 35](#_Toc410916315)

[Execution Model 35](#_Toc410916316)

[Immediate mode 35](#_Toc410916317)

[Command Syntax 36](#_Toc410916318)

[OpenGL Hierarchy 37](#_Toc410916319)

[Platform implementation 37](#_Toc410916320)

[OpenGL Context 37](#_Toc410916321)

[The steps of a typical OpenGL application 37](#_Toc410916322)

[Driver operations of OpenGL on Windows machines 38](#_Toc410916323)

[Open GL Revisions 38](#_Toc410916324)

[OpenGL 1.0 38](#_Toc410916325)

[OpenGL 2.0 38](#_Toc410916326)

[OpenGL 3.0 38](#_Toc410916327)

[OpenGL 3.1 39](#_Toc410916328)

[OpenGL 3.2 39](#_Toc410916329)

[Debugging OpenGL 40](#_Toc410916330)

[Program Architecture 40](#_Toc410916331)

[Debugger enabled architecture 40](#_Toc410916332)

[OpenGL ES 41](#_Toc410916333)

[OpenGL ES 1.X 41](#_Toc410916334)

[OpenGL ES 2.X 41](#_Toc410916335)

[Rendering Pipeline 42](#_Toc410916336)

[OpenGL ES SL (GLSL ES) 44](#_Toc410916337)

[Windows Systems/Libraries 45](#_Toc410916338)

[SFML 45](#_Toc410916339)

[SDL 45](#_Toc410916340)

[GLFW 45](#_Toc410916341)

[GLUT 46](#_Toc410916342)

[Chapter 4: WebGL 47](#_Toc410916343)

[Introduction 47](#_Toc410916344)

[WebGL Viewport 47](#_Toc410916345)

[HTML5 47](#_Toc410916346)

[The Canvas element 48](#_Toc410916347)

[Shaders in WebGL 48](#_Toc410916348)

[Vertex shaders 48](#_Toc410916349)

[Fragment Shaders 48](#_Toc410916350)

[Execution model of shaders 49](#_Toc410916351)

[Browser support 49](#_Toc410916352)

[Platform support 50](#_Toc410916353)

[WebGL Architecture 50](#_Toc410916354)

[Three.js 50](#_Toc410916355)

[Three.js environment 51](#_Toc410916356)

[WebGL 2.0 51](#_Toc410916357)

[Chapter 5: Improving performance on the web 52](#_Toc410916358)

[Introduction 52](#_Toc410916359)

[Asm.js 52](#_Toc410916360)

[Emscripten 53](#_Toc410916361)

[Native Client (NaCI) 53](#_Toc410916362)

[How it works 54](#_Toc410916363)

[Google ANGLE Project 54](#_Toc410916364)

[Chapter 6: Methodology & Design 56](#_Toc410916365)

[Research Question 56](#_Toc410916366)

[The Approach 56](#_Toc410916367)

[Design 57](#_Toc410916368)

[Risk Analysis 57](#_Toc410916369)

[UML Diagrams 57](#_Toc410916370)

[User Stories 57](#_Toc410916371)

[Test Plans 57](#_Toc410916372)

[VOPCs/ Class Diagram 57](#_Toc410916373)

[Benchmarks 57](#_Toc410916374)

[Metrics 58](#_Toc410916375)

[References 59](#_Toc410916376)

[Bibliography 61](#_Toc410916377)

# Literature Review

# Chapter 1: Introduction

This paper aims to investigate the functionality of OpenGL and WebGL, in order to analyse in what areas they differ and to what degree. Today web based applications can operate at speeds similar to desktop applications, with the addition of very detailed computer graphics to these applications; a rich user experience can be achieved. One area that this applies to in particular is video games, where speed and detailed graphics is essential. ***“With HTML5, web apps will now be able to rival the performance of native and desktop applications by providing a full desktop like experience on a website to your end users.”*** (Jakober, 2011)

OpenGL (Open Graphics Library) is a cross-platform API used for creating 2D and 3D graphics, it works with the GPU (Graphics Processing Unit), resulting in hardware accelerated rendering. OpenGL was created in 1991 by Silicone Graphics (SGI), now maintain by the Khronos Group. The OpenGL syntax is very similar the C programming language syntax. OpenGL is only concerned with rendering; its API offers no functions for windowing and input.

As the technology of mobile technology advanced and became more, an optimized version of OpenGL that featured more efficient functions and less data transference was developed that would allow these devices to render graphics. OpenGL ES, ES meaning *embedded systems*, is a cross platform API for graphics specifically designed for mobile phones, devices and consoles. It is targeted at systems that have low memory. It’s based on the OpenGL 1.5 specification. It’s optimized for performance resulting in lower battery power consumption. Android devices support the OpenGL ES API .

OpenGL ES 2.0 created in 2007, is based on the OpenGL 2.0 specifications, it features a fully programmable graphics pipeline over the fixed function rendering pipeline. Due to advances in web browsers and the introduction of WebGL, which allows graphics to be rendered within the web browser, web browsers had become less dependent on third party plugins. *“Today many of the things you previously had to use a plugin to run in a browser are now supported directly by the browser. In other words, plugins now replicate many of the same features that the browsers support (canvas rendering, sound, file access, databases, cache, etc.).”* (Jørgensen, 2012)

WebGL is a JavaScript based API based on OpenGL ES 2.0 specification. WebGL extends HTML5 by exposing the HTML5 Canvas element, using a context that links to the canvas elements to a access the computers GPU.  *“WebGL context serves as a binding between high-level JavaScript and low-level GPU operations.”* (Sloup, 2011)

All code is executed on the computers Graphics Processing Unit. WebGL gives developers full control of how graphics are rendered to the screen, this *control* is in the form of *Shaders*, which defines every vertex and colour property of geometries. Shaders run entirely on the GPU. *“Shader code is first stored as a string in a suitable variable. It is then stored in a shader object of the correct type and compiled. This has to be done for both the vertex and fragment shader. Then the two shaders are combined into a single program that the GPU can run to render”* (Conrad, 2014) *too long*

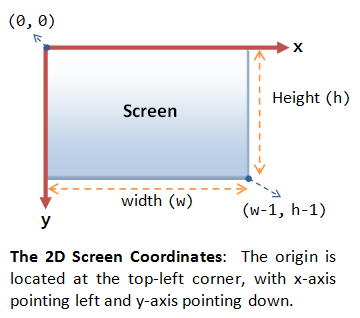
This paper will summarise how OpenGL and WebGL renders graphics and investigate the architecture.

# Chapter 2: Computer Graphics

Computer graphics is a field in computer science which is based on the creation of images and animations with the use of a computer. Computer Graphics dates back to 1960 it was pioneered by Ivan Sutherland. The first series of early computers consisted of nothing more than rows of blinking lights on a wall. Later when computers began to print data on paper, creative programmers would create ASCII art on sheets of paper. *“You might say that the first form of computer graphics was a panel of blinking lights. (This idea is supported by stories of early programmers writing programs that served no useful purpose other than creating patterns of blinking and chasing lights!) “* (Richard S. Wright, Haemel, Sellers , & Lipchak, 2013)

Later came the Cathode ray tube (CRT) which is the archetype all modern computer monitor. When formulas for lines and circles were published these graphics consisted of very simple 2D and 3D lines could be drawn on a screen. Today all computer screens are raster based and composed of pixels, each defined by colour (R, G, B) and Positon (x, y).

*“The origin (0, 0) is located at the top-left corner, with x-axis pointing right and y-axis pointing down. This is different from the conventional 2D Cartesian coordinates, where y-axis is pointing upwards.”* (Hock-Chuan, 2012)



## Applications

There is a vast array of applications within the field of computer graphics

* Computer aided Design (CAD)
* Videos games and animated movies
* Data Visualization
* Medical Visualization

## GPU

The graphics card or Graphics Processing Unit or Visual Processing Unit is used for processing triangles and converting it into pixels within a framebuffer at very fast speeds. The processor is optimized for 2D/3D graphics. It is a highly optimized, parallel, multithreaded multiprocessor device. In fig1.0, is a magnified image of the GPU. It is more efficient at processing graphics than the CPU due the highly parallelized architecture of the device. The GPU is equipped to process millions of vertices and fragments within a second. This division of labour leads to less processing of complex operations, which lessens the time required to process vertices, fragments, etc.

## CPU

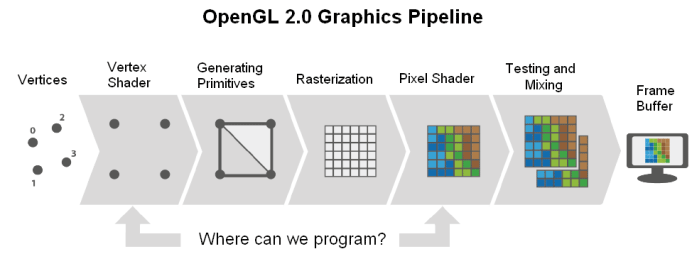
The CPU (Central Processing Unit) is used the run to the operating system and applications of the computer. *“As a general rule, the CPU is used to carry out large and complex operations, whereas the GPU is excellent at handling large numbers of simple tasks. When choosing which approach to use, it depends on the task at hand and the amount of data to be processed.”* (Tchernitski, 2013)

## Designing efficient applications

To create an efficient application, it must be structured so that the workload is even distributed between the CPU and GPU. Areas of overhead must be identified, for example. The transference of data between the CPU and GPU is done via *buses* these buses operate at slower speeds that adds overhead to the performance of OpenGL applications. *“To get great performance in your application, you must carefully design your application to feed data and commands to OpenGL so that the graphics hardware runs in parallel with your application. A poorly tuned application may stall either on the CPU or the GPU waiting for the other to finish processing.”* (Apple Inc., 2012)

# Modern Rendering pipeline

Every 3D scene viewed on a computer screen is generated as the result of a series of processes and successfully tests that were computed by the GPU which is referred to as the rendering pipeline.

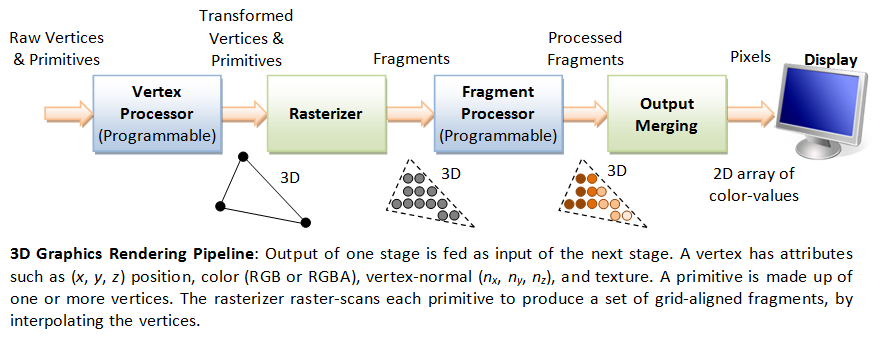


The rendering pipeline consists of a series of stages required to render graphics to the screen. Within the pipeline once a stage is completed the output of that stage is sent to the next preceding stage, some stages work in parallel to each other, but ultimately this procedure is continued until the output reaches the framebuffer, where the output is rendered to the screen. *“Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages.”* (Segal & Akeley, 2004)

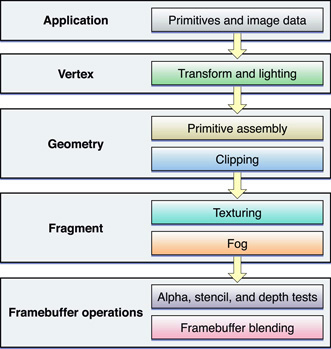
*“Because of this pipeline architecture, today's graphics processing units (GPUs) perform billions of geometry calculations per second. They are increasingly designed with more memory and more stages, so that more data can be worked on at the same time.”* (Definition of:graphics pipeline, 2001)

OpenGL is used for rendering graphics to the screen, by acquiring data and processing it into something aesthetically pleasing, this process is referred to as *rendering*. The rendering process can be greatly accelerated using specialized hardware. But the majority of OpenGL pipeline operations are software based. The user is oblivious to where the software and hardware operations begins and ends, but it is specified in the OpenGL specification.

In general, the application sends vertices that eventually form geometries through the GPU within the GPU the frame buffer creates the final processed image. The vertex processing stage is where each vertex is processed to determine the location of its corresponding geometry in the framebuffer. Once this stage is completed the rasterizer checks the pixels which are affected by geometry. The fragment processing stage determines the final colour of each pixel.

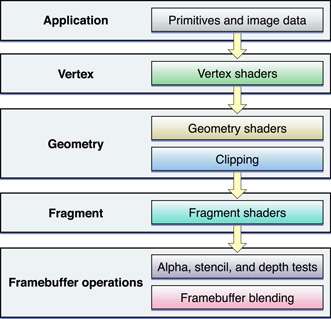


The initial pipeline in OpenGL 1.0 consisted of the following stages.



## Modern OpenGL

Due to advances in technology the fixed graphics pipeline is now obsolete. Today the graphics pipeline is *fully programmable* it is also referred to as *Modern OpenGL* which was introduced in OpenGL 2.0. “Programmable pipeline” means that the internal mechanisms can be modified. This modern implementation of OpenGL is completely *shader* based. Older versions of OpenGL implemented a *rasterization based graphic pipeline*, and fixed function processing. These functions/processes are now deprecated due to advances in technology. The below diagram illustrate the revised graphics pipeline with the inclusion of shaders.



## GL Shading Language

GLSL (OpenGL Shading Language) is a shading language used for rendering of realistic real time computer graphics, for example computer games. Developed by the ARB, it defines shaders that are small programs that execute on the GPU. Shaders allows for increased control over the graphics pipeline. It was added as an extension to OpenGL 1.4, and then added to the core specification in OpenGL 2.0. GLSL is simply a series of strings which are passed to the GPU. GLSL is the shading language used for OpenGL, OpenGL ES 2.0 and WebGL. All programs written with GLSL must have one main function. GLSL supports user defined functions but these functions must have a return type.

### Control flow within GLSL

GLSL supports simple but effective control flow.

* if-else statements
* for, while and do-while statements

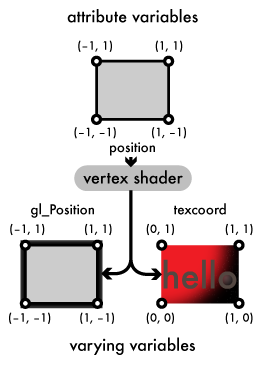
### Shaders

Shaders allow a programmer to modify the operations of the graphics pipeline. Essentially shaders are what define the modern graphics pipeline. Shaders define how vertices and materials and lights appear on a screen they carry out complex calculations to render pixels. In the old graphics pipeline vertices where passed through a model view and projection matrix, and then outputted to the correct window coordinates and finally defined by a colour. With the modern this entire process must be defined by shaders.

Shaders programs are written in GLSL which has syntax similar to the C language and are compiled on the GPU. Introduced in OpenGL 1.4 and then added to the core specification in OpenGL 2.0. Shaders are typically very small programs but can be as large as several hundred lines of code based on its complexity. The main advantage of shaders is that they give programmers complete control over every vertex. A complete shader program is composed of two smaller distinct shaders a vertex and fragment shader both of these shaders are again referred to as a shader.

#### Vertex Shader

This shader defines the vertex position using gl\_Position. It outputs a clip space position in the form of a varying value to the rasterizer. The below diagram illustrates gl\_Position defining the attribute variables

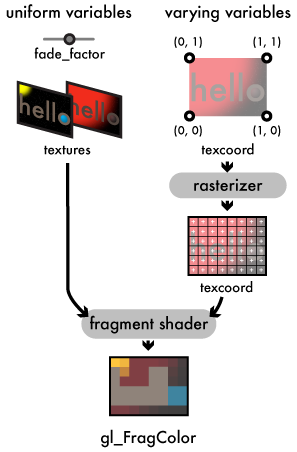


##### Syntax



#### Fragment shader

Fragments are sent to the fragment shader, where lighting and texture mapping occur. Typically this part of the pipeline is prone to bottlenecks due to the complex operation that occurs. This shader outputs a colour and depth for a fragment, which are received by the frame buffer. Varying values within the fragment shader correspond to another varying value within the vertex shader, gl\_FragColor outputs the colour from the fragment shader. Below diagram illustrates the gl\_FragColor.



##### Syntax



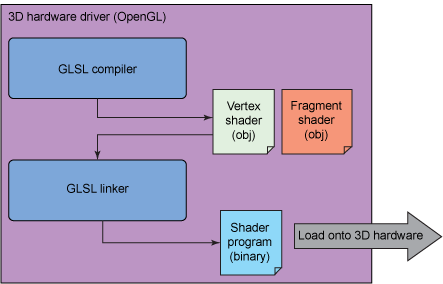
Once both of these two mini shaders are compiled using ***glCompileShader()***, they are merged together ***glLinkProgram()*** to form a executable ***Program*** using ***glCreateProgram()***. Shaders can be written as a string with C syntax or stored within a text file, accessed via ***glShaderSource().***

### Procedure to create a program

* GLuint prog = glCreateProgram();
* glAttachShader(prog, vertex\_shader);
* glAttachShader(prog, fragment\_shader);
* glLinkProgram(prog);

### Execution model of shaders

The driver contains a compiler and linker for generating complete shader programs. First the vertex and fragment shader are compiled into binary code by GLSL compiler. Then the GLSL linker “links” the vertex and fragment shader to form a full shader program which is referred to as a Program, this program will then be executed on the GPU. Figure 7 illustrates the process of operations for generating a shader program within the driver.



### Communication between shaders

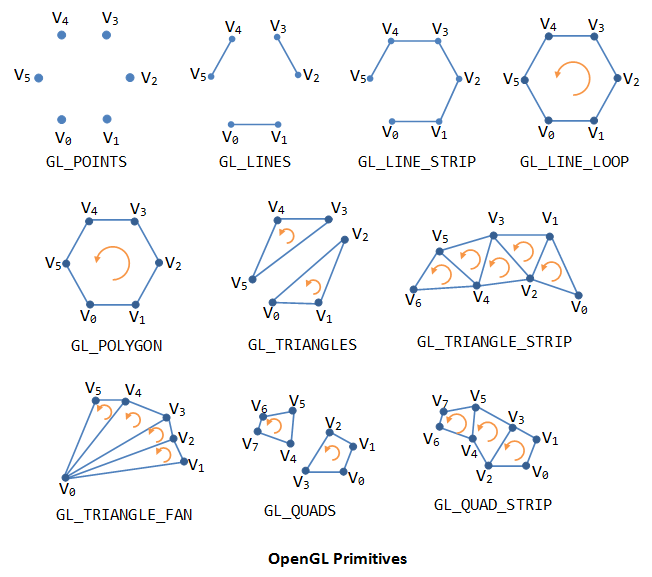
Shaders acquire vertices as inputs which consist of having either one or more attributes. The OpenGL API permits that additional variable can be applied to a vertex to complete shading. These variables are *attributes* and *uniforms*. Attributes can only be read but not written by the vertex shader and are defined on a per vertex basis, attributes can vary from vertex to vertex. Uniforms are perceived as constant global variables that can be applied to all vertices examples of uniforms consist of light intensity and gravity.

Additionally once a vertex is processed by the vertex shader it is ready to be assembled into a primitive. This primitive consist of fragments that contains interpolated variables, each fragment is then sent to the fragment shader. These interpolated variables are referred to as *varying variable* or *varyings*, which are necessary for computing the value per fragment via reading interpolated data.

## Geometry Stage

All geometric objects are defined by vertices (x,y,z,w), a vertex is a series of data related to a single point in space. A vertex has attributes such as position, colour and textures coordinate. *“The type of properties provided per vertex include x-y-z coordinates, RGB values, translucency, texture, reflectivity and other characteristics.”* (Definition of:graphics pipeline, 2001)

These vertices can be in the form of a *point, line, triangle*, etc. All primitives are composed of one or more of these vertices, listed below are the ten primitives featured in OpenGL.



The vertices of a geometric object are stored within a VBO (vertex buffer objects) multiple VBO’s can be associated with one geometric object. These VBO’s are stored within a VAO (vertex array objects) the purpose of a VAO is to organize multiple VBO’s.

### Vertex Specification

The initial process is to prepare a series of vertices for rendering for the next stage within the graphics pipeline. A stream of ordered vertices is created these vertices represent the edges of a primitive. A primitive is typically a shape, for example points, lines and triangles, etc. The vertex specification process primary deals with the *Vertex Array Objects* and *Vertex Buffer Objects*. Vertex Array Object (VAO) defines the data a vertex contains, Vertex Buffer Objects (VBO) defines the attributes about a vertex. These *vertex attributes* are eventually passed to the *vertex shader*. Each of these attributes is a vector, consisting of four floating point numbers.

**ARB\_vertex\_program**

## Vertex buffer objects (Buffer Objects)

**GL\_ARB\_vertex\_buffer\_object**

The purpose of vertex buffer object (VBO) is to improve the performance of OpenGL. It holds the *vertex attribute data* for shaders. It features the benefits of *vertex array* and *display list*. Vertex array data is stored within the high performance graphics memory.

A buffer which stores pixel data is refer to as a *Pixel Buffer Object* (PBO). The main benefit of the vertex array is that it reduces the amount of function calls. It creates *Buffer objects*, which are stored within the high performance memory on the server side. Also vbo is server side, resulting in the buffer objects having the capability of being shared on many clients.

## Vertex Array Object

The Vertex Array Object holds information about vertex data and *vertex buffer objects*. This information is shared amongst the clients.

## Vertex Processing

This step maps a vertex to a 2D co-ordinate onto the screen. It also applies a transform the vertices using one or many shaders.

#### Vertex Shader

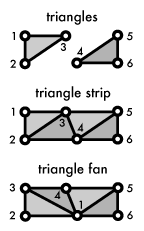
The GPU sends a vertex from the vertex array to the vertex shader. The shader then outputs attributes which are referred to as ***varying*** values. The function of the vertex shader is to determine the coordinates of each vertex within the screen. Additionally it also determines the light intensity and colour of that vertex.

#### Triangle assembly

The vertices are then connected together to form triangles these vertices are ordered in the same way they’re ordered in the element array.

A *triangle strip* is defined, by reusing the previous two vertices of a triangle, as first two vertices of the next triangle.

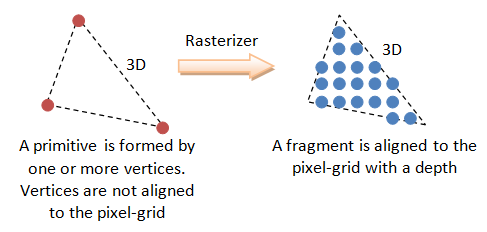
A *triangle fan* connects the first elements of each triangle together.



## Rasterization

Rasterization is the actual drawing of the pixels from vertices. It is the process of converting objects to pixel colours within the frame buffer. Rasterisation goes from a pixel value within the world coordinates to a pixel value in the screen coordinates. This is also referred to as the pixel pipeline. It acquires a series of vertices of an object, it then transforms each of those vertices into an array of 2D pixels, that array Is sent to the monitor and it then fills the outputted 2D triangle.

A primitive is defined by a set of one or more vertices these vertices constitute the area of a primitive. This area is *raster-scanned* after this scanning process is completed a series of fragments is created, that represents the area of the primitive.



Fragments are interpolated from vertices and are very similar to pixels, but what differentiates them from pixels is that they also have properties of vertices, such as normal and texture. Fragments are grid aligned.

The rasterizer converts these points into a group of fragments, fragments are similar pixels except they can store more information for example Normal, Textures co-ordinates and a Depth value. *“At this point, the triangle are now defined in terms of coordinates on the screen. The x and y coordinates define the x and y position of the vertices on the screen and the z coordinate defines the depth of the vertex (how far in the vertex is).”* (Mallet, 2014)

### Viewport transform

The viewport is a rectangular region located within an applications window that is used to display graphics. The viewport’s region is measured in pixels, its origin being at the top-left corner. The size of the viewport can be defined using the ***glViewport()*** function. To correctly configure a scene a projection plane must be defined. These are defined using ***gluPerspective()*** or ***glOrtho()***. Both the viewport and projection plane must have the same *aspect ratio*, otherwise the scene will be inaccurate.

### Back-face culling

This process removes triangles that are behind other triangles that are visible on the screen. This process removes triangles that are not facing the camera. This culling is issued using glCullFace(). It takes two forms “front-face” and “back-face”, each can enabled separately for example glCullFace(GL\_FRONT), or both can be enabled glCullFace(GL\_FRONT\_AND\_BACK).

### Frustum culling

The *view frustum* represents the volume of space that contains every piece of geometry that can potentially be rendered to the screen. The view frustum exists between the *near plane* and *far plane* essentially it is a pyramid without a point, this is illustrated below.

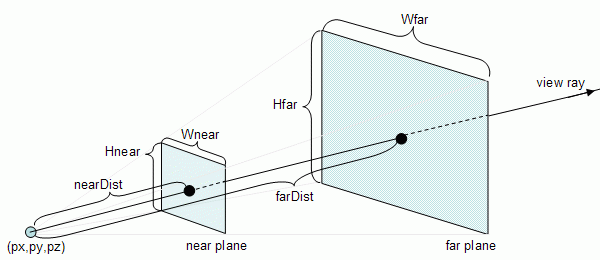


Fig Conceptual model of the near and far plane.

The GPU only requires frustum culling to determine whether geometries are present within the view frustum. This helps the GPU performance greatly by not processing unseen vertices.

## Per-Fragment Processing

Per fragment processing is a series of tests and operations that are applied to a fragment before the values of its component are copied to a corresponding pixel located within the frame buffer, once the series of test are completed. If one of these tests fails the fragment will be discarded, this processing focuses on texture and lighting

### Pixel Ownership Test

This initial test determines if the pixel within the framebuffer and its associated fragment are both governed by the OpenGL context. If this test is passed, the scissor test is enabled. If it fails, the window system determines if the fragment is removed. An example of this occurrence is when another GUI, hides an OpenGL window.

### Scissor Test

Test/Clips a fragment against a rectangular region. This is implemented using the glScissor() command.

### Alpha Test

This test determines if whether to remove a fragment based on it is alpha component value, using glAlphaFunc()

### Stencil Test

The purpose of the stencil test is to remove a fragment from the graphics pipeline. The test is based on comparing the stencil value of two fragments, one is the current stencil value and the other within the stencil buffer. If the test fails the fragment is removed. The function ***glStencilFunc()*** outlines a test to be carried out on each pixel in the stencil buffer. Once this function is completed the function ***glStencilOp()*** is called, which specifies an operation to do based on the result of the test.

### Depth Test

This test determines whether one fragment occludes another, based on the Depth values of both fragments. If the depth value of the current fragment is greater than new fragment, the newer one is removed.

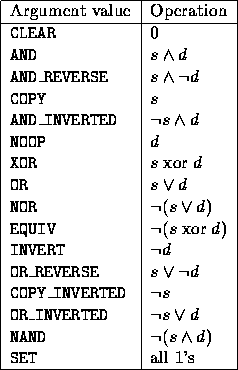
Alpha blended This stage blends the components of a fragment with another located in the framebuffer. The actual blending is dictated using the glBlendFunc() function.

### Dithering

Dithering is required to select a colour to display on a screen, that both match what the screen is capable of displaying (low precision colour) and what the actual calculated (high precision) colour. In OpenGL, dithering refers to an algorithm that’s based on a fragments colour. The algorithm processes a fragments colour, x and y coordinates. Dithering is the only operation enabled by default.

### Logical Operations

Is a per-sample operation targeted at both a fragments colour value and the equivalent colour value within the framebuffer. These operations are only based on colour indices. These operations operate independently for each colour index. The calculated result updates the frame buffer value. Logical operations are selected using the ***glLogicOp()*** function. This function requires one enumerated argument, a list of the available arguments are listed below. Also the ***s*** represents the current fragment value, and ***d*** represents the framebuffer value.



### Write Mask

The write mask governs if a component (colour, depth, or stencil) is written to the frame buffer.

#### Colour Mask

The fragment shader outputs colours which can be masked, also the components (R, G, B and A) can also be masks.

#### Depth Mask

The depth mask is typically used for issuing transparency within a screen. When this mask is enabled the depth buffer is not updated.

#### Stencil Mask

The stencil mask requires an integer argument rather than a boolean argument like the other masks. This mask generates a bit mask used either the sending bits to the stencil planes.

## Output Merging

Z buffer is used for hidden surface removal the z buffer can be initialized via the GLUT. glutInitDisplayMode(GLUT\_RGBA | GLUT\_DEPTH) and to enable to it the glEnable(GL\_DEPTH\_TEST) command is used.

## Framebuffer

The framebuffer is final stage of the graphics pipeline it is located within an area in memory. The framebuffer is a collection of buffers. The *graphics accelerator* is used for drawing the actual graphics to the screen, it also has section in its memory dedicated to updating every pixel within the screen, and these pixels are bytes in the graphics accelerator.To prevent the screen from flickering *display memory* is used to maintain the pixels on the screen, the display memory is updated many times per second.Another section of memory is dedicated to storing data that is not visible on the screen this section is referred to as *offscreen memory.* The *Graphics context* is governs the operations of the rendering pipeline. It dictates the state to use, and manages the memory required to draw graphics to the screen.

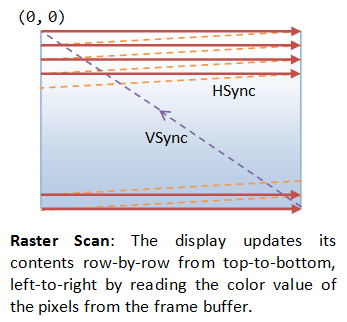
The frame buffer is composed of the following buffers

* **The Colour buffer**
* **The Depth buffer**
* **The Stencil buffer**
* **The Accumulation buffer**
* **The Multisample buffer**
* **The Auxiliary buffer**

### Double buffering

*Double buffering* ensures no half rendered images are displayed this means that two buffers the *front buffer* and the *back buffer* working in parallel in order to display a rendered frame to the screen. The front buffer renders what is visible to the screen while the back buffer renders what is not visible on the screen. Once the rendering process is complete by the *front buffer* the contents of the *back buffer* are copied over into the *front buffer* the both buffers are swapped this operation happens for every frame.

Additionally during the swapping process between the buffers a *vertical synchronization (VSync)* which is the synchronizing of this swapping procedure with the refresh rate of the monitor, ensuring that the frame rate doesn’t exceed the refresh rate of the monitor. This operation occurs when the next swapping of buffers is ready the GPU stalls until the next VSync to carry out the next swap.



### Depth buffer

The *depth buffer* stores the depth value a pixel. The depth refers to the Z component, it is the distance measured that is perpendicular to the screen. It is required for hidden surface removal. When an additional pixel is displayed, a depth calculation is implemented to determine whether the additional primitive is drawn or not. The depth buffer is primary used when working with 3D primitives.

### Stencil buffer

The *Stencil buffer* is a section of memory used to test and to mask sections of a complex shape and it determine whether to update each pixel.

### Accumulation buffer

The *Accumulation buffer* uses high precision components. It takes several images and stockpiled them to produce a composite image.

### Multisample buffer

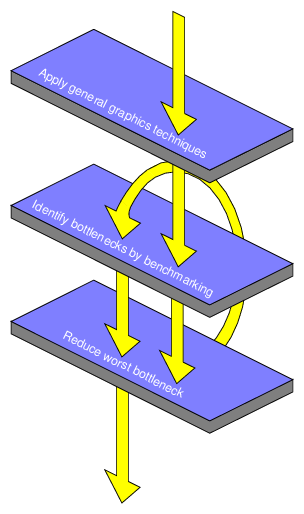
The *Multisample buffer* is a section of memory that operates by gathering different samples. It then averages the differences between these pixels, to eventually produce an antialiased image.

### Auxiliary buffer

The *Auxiliary buffer* are off screen memory buffers which stores data from calculations to be by other processes. A typical frame buffer would have many auxiliary buffers

## Bottlenecks

The function of the graphics pipeline is render images as fast and efficiently as possible the graphics pipeline is only as fast as it slowest part. The slowest part of this pipeline is referred to as a bottleneck. Trying to improve the rendering performance of computer graphics applications requires identifying bottlenecks within the pipeline by benchmarking distinct parts of the graphics pipeline, then identifying what process is taking too much resources and time to complete. Once identified the programmer modifies the process to optimize the performance of the graphics pipeline. This is illustrated in the below diagram.



# Chapter 3: OpenGL

## Introduction

OpenGL (Open Graphics Library) is a 2D API (Application Programming Interface) used for creating highly detailed applications via hardware accelerated rendering. It is desktop centric API meaning it’s not used on mobile devices, its operating system and window system independent meaning that it highly portable across varied computing environments. It’s designed to be very close to the hardware thus allowing for excellent performance. It’s only focused on rendering.

## Background

Before OpenGL, writing software that could operate on the majority of systems was a difficult process as there many different graphics hardware devices and each had its own configuration. Programmers where required to create distinct driver functions for each these devices. In the early 1990’s, Silicone Graphics Inc. (SGI) was the leader in the field of computer graphics due to the simplicity of their “IRIS GL API”. The IRIS GL (Integrated Raster Imaging System Graphics Library) was used for rendering 2D and 3D graphics on the desktop. When its other competitors (IBM, Hewlett-Packard, etc.) began to delve into the field of computer graphics the influence of SGI on the computer graphics market fell which led SGI to turn their IRIS API into an “open standard” whence know as OpenGL.

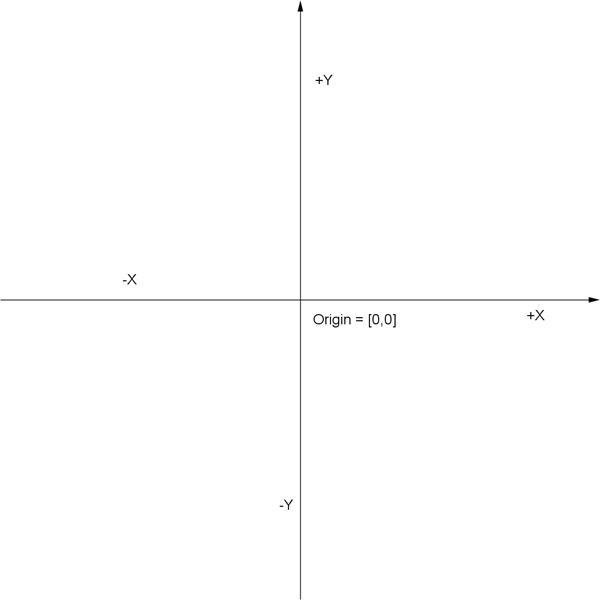
OpenGL was developed in 1991 by Silicon Graphics Inc. and released in 1992. OpenGL was originally maintained by Architectural Review Board (ARB). This board consisted of SGI, Microsoft, NVidia HP, 3DLabs, IBM etc. this group controlled the specification of OpenGL. *“The specification for this API was finalized in 1992, and the first implementations appeared in 1993. It was largely compatible with a proprietary API called Iris GL (Graphics Library) that was designed and supported by Silicon Graphics, Inc. To establish an industry standard, Silicon Graphics collaborated with various other graphics hardware companies to create an open standard, which was dubbed ‘OpenGL.’”* (Section 1.1 OpenGL History, 2010)

In 2006 the ARB decided hand over the OpenGL API to the Khronos Group. Founded in 2000 it’s a non-profit, Oregon based organization dedicated to maintaining and improving computer graphics and parallel computing.

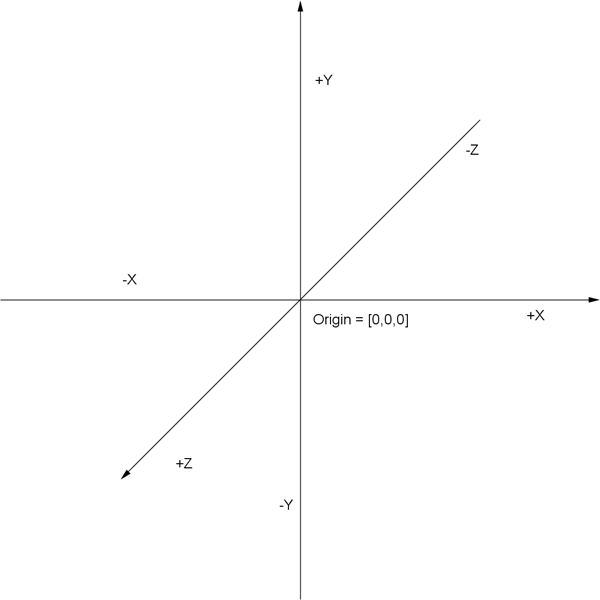
## Fundamentals

### Coordinate system

All computer graphics APIs use the *Cartesian coordinate system* to represent a two dimensional environment which is composed of and x and y axis. The x represents the horizontal axis while the y represents the vertical axis with the point of origin located where both axis intersect. This is illustrated in the below diagram.



To create a three dimensional environment an additional axis is required. The z axis is perpendicular to both other axis’s it also intersects at the origin. Illustrated in the below diagram.

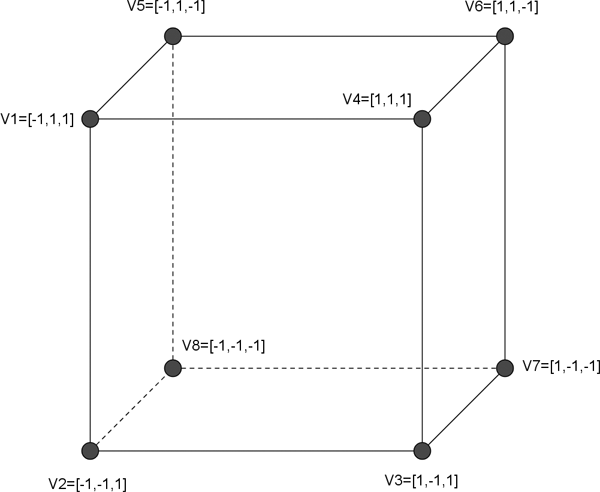


In the realm of computer graphics several additional coordinate systems are implemented to correctly position and modify all the components of a 3D scene. These include

* Object Space
* World/Model Space
* Camera/View Space
* Screen/Clip Space

In order to modify an object within a 3D scene a mathematical object referred to a matrix is used. In the next section of this chapter a more in-depth account of matrices is given.

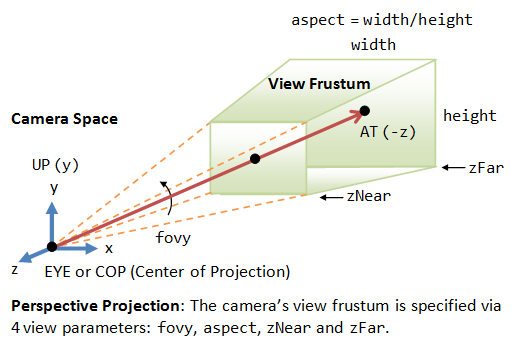
Object Space relate to the local coordinate system position of a 3D object which is the object's vertices relative to the object's origin.



**World/Model Space** relates to defining the positions for all objects within a scene to position the above cube within the world space, the object’s coordinate system position is moved to the World coordinate system position. This is achieved using a *world matrix* that is applied to each vertex of the above cube within object space.

**Camera/View Space** relates to a camera that looks at the scene. Each vertex within the world space coordinate system is multiplied by a *view matrix*. The camera is positioned on the z axis and is aimed towards the negative section of the z axis.

**Screen/Clip space** is the transformation of a three dimensional space (the 3D scene) into a two dimensional space (the screen). The vertices within the camera space are multiplied by a projection matrix the result of this process is what is viewed on the screen.This transformation is achieved using either an orthographic projection or perspective projection within an orthographic projection all objects appear to have the same size despite their distant from the viewer. While the perspective projection is just how the world is viewed from a person’s eye, for example a cow that appears is small, is only small because it is far away. Below is an illustration of perspective projection.



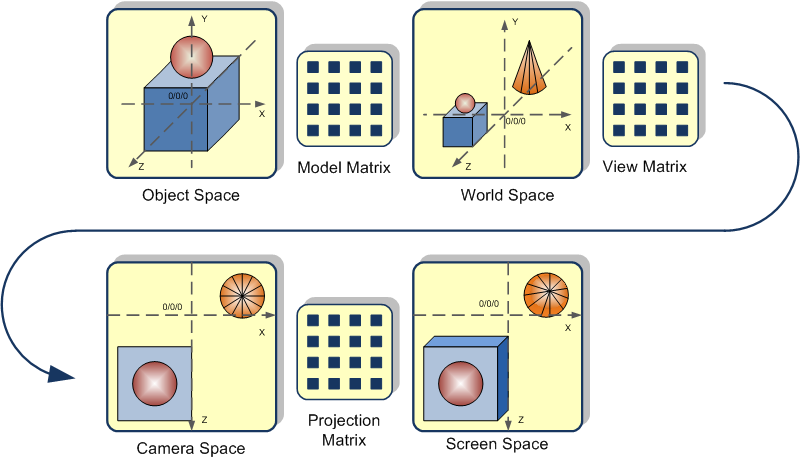
Perspective projection is enabled with the below line of code.

**gluPerspective(fovy, aspectratio, near, far);**

* *fovy* is the angle of the field of view.
* *Aspect ratio* is the aspect ratio of the frame.
* *Near* and *far* are the distances between the viewpoint and the clipping planes.

Below is a simplified illustration of this process

* Vertices a cube are defined within object space
* These vertices are moved from object space into world space via a model matrix.
* The vertices now in world space are multiplied by a view matrix to put them in the view space (camera)
* The view space vertices are multiplied by a projection matrix to be placed within the screen space.



### Normalized device coordinates

Once all the transformations are completed the vertices coordinates should be in normalized device coordinates. Normalized coordinates range from -1 to +1. The bottom left (-1, -1), and the top right (1, 1).

### Transformations and Matrices

Geometries are composed of many vertices and each of these vertices has a position to modify geometry in any way, would involve applying a calculation to each vertex which very inefficient process. In order for geometry to move, change its orientation or size efficiently within a 3 dimensional coordinate world a transformation by a matrix needs to be applied to it.

Transformations are the technical term for modifying geometry, without applying a calculation of each vertex. Translations pertain to the moving of geometry’s position. Scaling relates to modifying the size of geometry and rotation simply relate to changing the direction geometry is facing. All transformations are based on a matrix.

A matrix is an object containing a series of elements that applies calculations to the vertices of geometry typically a matrix is represented by a 4X4 based matrix. This section will give a high level description of matrices, but not in too much detail as it is beyond the scope of this paper. Within OpenGL matrix operations are carried out by multiplying the matrix by a vector.

#### Translation

The translation matrix is the most easy to understand the x, y and y refer to values to be added to the current position.

#### Scaling

A scaling matrix is scale an object to either a larger or smaller size.

#### Rotation

A rotation matrix is used to rotate an object around a specified axis, the below matrix rotates an object around X axis.

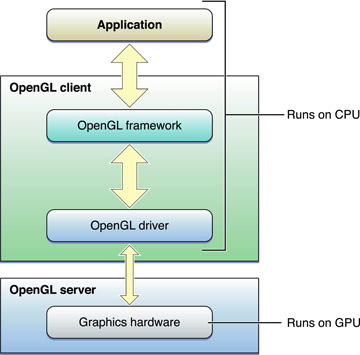
The below matrix rotates an object around Z axis.

The below matrix rotates an object around Y axis.

## Open GL Architecture

The architecture of OpenGL is based on a Client-Server model. The *Client* is the actual application program that issues drawing commands (functions), the *Client* runs on the CPU. The *Server* is the graphics engine that consists of the shader programs. The Server receives commands issued from the client these commands are processed in the order they are received, and performs the drawing operations, and the server runs on the GPU. A server can have many GL *contexts* each of these contexts is an encapsulated GL state. A client can access any of these contexts.

Processes within OpenGL architecture are referred to either being server-side or client-side. The actual *rendering process* is based on the CPU and GPU working together. When designing this *rendering process* dividing the workload efficiently between the CPU and GPU is critical, to developing efficient systems. Additionally how data is stored and transferred amongst the GPU and CPU is equally important.



### OpenGL architecture within a network environment

Within a networked computer environment, for example an office building, college campus or Local Area Network (LAN) a series of computers is connected to each other. The OpenGL architecture has the capabilities of operating within these networks, by dividing the workload of the rendering process amongst multiple computers within the network. OpenGL is network-transparent this is also referred to as indirect rendering, it implement the GLX extension on the X Window System (when using Linux). It typically operates via the X server. All graphics data are transported over the X11 protocol. In order for indirect rendering to perform efficiently the network speed must be very fast typically within the Gigabyte Ethernet range, and if the application doesn’t modify any of the rendered geometry. The computer additionally requires a fast 3D graphics accelerator.

One or more computers are used to display the rendered graphics, these computer(s) are referred to as the *client*. While another computer (*server*) is used to compile the graphics program. The format for transmitting OpenGL commands (called the protocol) from the client to the server is always the same, so OpenGL programs can work across a network even if the client and server are different kinds of computers. Resulting in OpenGL applications within a network the client and server are not the same type of computer.

## Execution Model

A context state controls a series of shader programs which creates primitives, which OpenGL eventually draws. OpenGL operates by rendering data into the framebuffer, and reading data within the framebuffer.

A primitive can be a point, line or even a polygon. Primitives are composed of one or more vertices. Each vertex is defined by a position, texture co-ordinate, normal etc. Each vertex is processed independently. *“Commands are always processed in the order in which they are received, although there may be an indeterminate delay before the effects of a command are realized.”*

### Immediate mode

The first iteration of OpenGL’s used to draw primitives was very simple this was refered to as immediate mode or the Begin/End paradigm. It consisted of *glBegin()* and *glEnd()* commands in order to display primitives to the screen. Within the *glBegin()* and *glEnd()* commands the vertices and arrays are drawn, using the *glVertex()* and *glDrawArray()*. The following code segment features the immediate mode.

glBegin(GL\_TRIANGLES);

    glVertex3f( 0.0f, 1.0f, 0.0f);

    glVertex3f(-1.0f,-1.0f, 0.0f);

    glVertex3f( 1.0f,-1.0f, 0.0f);

glEnd();

With these commands pixels for a polygon would be stored within the system memory and be displayed to the screen. Immediate mode is obsolete because, the graphics card connects to the program flow. The driver doesn’t inform the GPU to begin rendering before the *glEnd()* call. Due to the fact it doesn’t know when to stop sending data. Additionally it had too many API calls. Due to advances in hardware technology, *glBegin()* and *glEnd()* are now obsolete, the *vertex buffer Object* (VBO’s) is what is currently used by the OpenGL programs. There’s a noticeable performance improvement when compared to immediate mode, because the data is now stored on the *video device memory* resulting in the *video device* directly rendering it.

## Command Syntax

OpenGL has a specific convention when writing functions

C:\Users\James\Desktop\vvvvvv.jpg

* **gl** means it belongs to the OpenGL API
* **Colour** sets the current colour
* **3** represents the amount of arguments
* **f** letter represent the type (d: double, f: float, ub: unsigned byte)

## OpenGL Hierarchy

GL, (commands for vertex and matrix manipulation)

GLU, GL Utility Library (consist of Helper functions)

GLUT, the GL Utility Toolkit (Windowing system)

## Platform implementation

It has a platform specific implementation for both Windows and Linux systems.

* Windows (opengl32.lib, glu32.lib, glut31.lib)
* Linux (-I GL, -I GLU, -I GLUT)

## OpenGL Context

The OpenGL context is a state machine that maintains information relating to the rendering process of the current instance of OpenGL. An OpenGL application cannot exist without a context. A context can be a distinct plane/surface, for example a window within a windowing system. In order for any OpenGL commands to work, a context must be *current*; all OpenGL commands affect the state of whichever context is current. OpenGL commands can operate on a context, only when it is current.

### The steps of a typical OpenGL application

1. A window is defined.
2. An OpenGL context is defined.
3. The context is linked to the window and current process.
4. Switch to created context
5. Render primitives
6. Destroy context
7. Destroy window

## Driver operations of OpenGL on Windows machines

On Windows based systems, when the OpenGL application is compiled its links with the “opengl32.dll” (a driver), opengl32.dll is then loaded. The registry is checked for any GL drivers. If a GL driver is present it is then loaded. Opengl32.dll uses OpenGL 1.1 operations.

## Open GL Revisions

### OpenGL 1.0

The first version operated with a *fixed function pipeline*, meaning that everything in OpenGL was fully defined. No part of the sequence of operations could change, and only the values of positions and colours for vertices could be altered. OpenGL continued to improve upon its fixed function pipeline, but it wasn’t until advances in GPU’s technology, allowed for further improvements to OpenGL graphics pipeline specification. The fixed function pipeline remained until version 2.0 (2004).

### OpenGL 2.0

OpenGL 2.0 introduced the concept of programmable shaders. Shaders are simply small programs that can manipulate the pipeline; they’re broken into two distinct groups.

* *Vertex shading*: allowing for control of geometry/meshes.
* *Fragment shading:* allowing for shading of a pixels colour.

OpenGL 2.0 is backwards compatible, allowing for both the fixed and programmable pipeline to be used. The advent of OpenGL 3.0 would introduce non backwards compatibility with previous version of OpenGL. It also introduced fragment operations

### OpenGL 3.0

OpenGL 3.0 introduced the ***depreciation model*** which removed the older features of OpenGL. This decision was made by ARB of the Khronos group. While the depreciation model stopped supporting older versions. It did ultimately improved performance and minified the specification.

It also changed how context operated. A context is used to store shaders and information. Contexts are categorized into two groups.

* ***Full context***: Implements features of the used OpenGL version, including the depreciated features
* ***Forward compatible context***: Only have features that will be available with the next version of OpenGL.

Forward-compatible contexts are now included in every version from OpenGL 3.1

### OpenGL 3.1

Two of the major changes to this revision were.

* The fixed function pipeline was removed.
* Data would now be stored in the GPU’s buffer objects.

With all a new changes to the architecture, the ARB decided to add new extensions that allowed depreciated features to be used.

**GL\_ARB\_ compatibility**

### OpenGL 3.2

A *geometry shader* was added which allowed for geometry to be created and modified from within the pipeline.

*Profiles* were also introduced, which gave developer better selection of which suite of features to use. Essential these Profiles categorized the contexts into two groups, core and compatible.

* ***Core***: is a minified and has the latest features of OpenGL version
* ***Compatible***: features every functions of every version of OpenGL up to the existing release.

## Debugging OpenGL

Debuggers are used to analysing performance and profiling applications. OpenGL applications works by linking to an OpenGL library, this library communicates to the graphics card via the drivers. There are vast arrays debugging tools that offer gather and output the software metrics.

### Program Architecture

When using a debugger tool, the debugger is inserted between the OpenGL program and the OpenGL library. The debugger acts by capturing the “calls” from the OpenGL program. Then it executes the same “calls” on the OpenGL library.

### Debugger enabled architecture

Primarily the debugger is used for:

* finding errors
* Logging data
* Measure memory usage
* CPU and GPU performance

# OpenGL ES

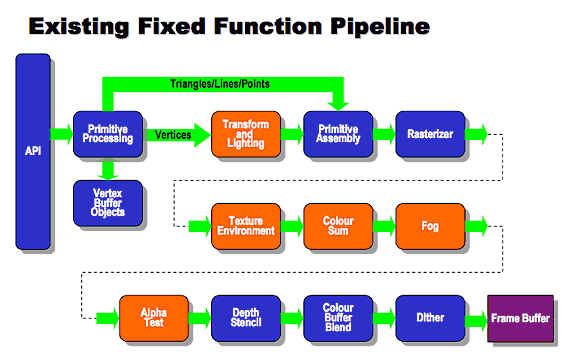
It is designed for embedded systems. It has been adapted to remove redundant calls and functionality.New features added for embedded systems. It is best API for 3D handheld games.Based on OpenGL 3.1

## OpenGL ES 1.X

Designed for fixed functions and improved performance.

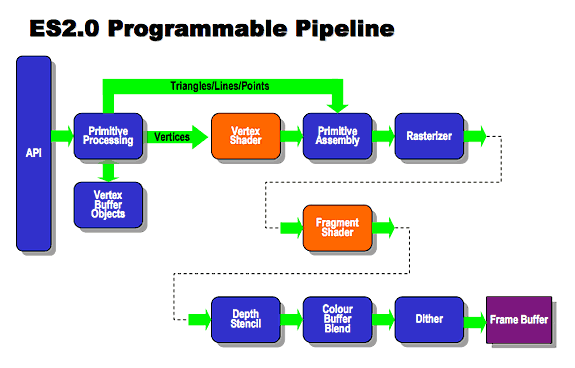
OpenGL ES 1.1 is based on OpenGL 1.5

Added 3D graphics API for android and Symbian

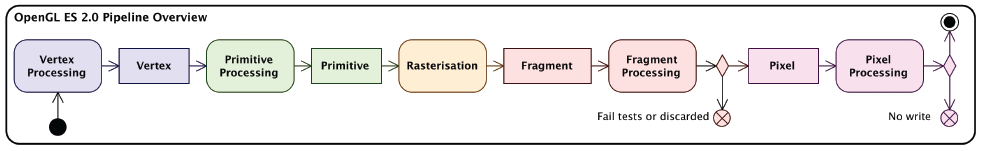


## OpenGL ES 2.X

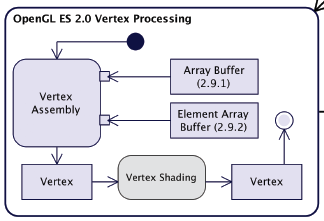
It is similar to the OpenGL 2.0 specification. It features a fully programmable pipeline. It features the ability to create vertex and fragment shaders. It does not support fixed function pipeline. The programmable pipeline is illustrated below. It only features shaders is not backwards compatible with 1.x



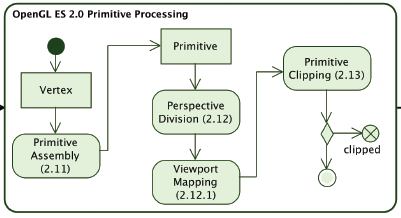
### Rendering Pipeline

**

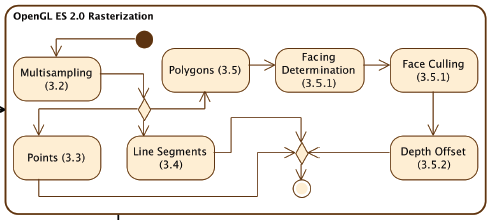
#### Vertex processing

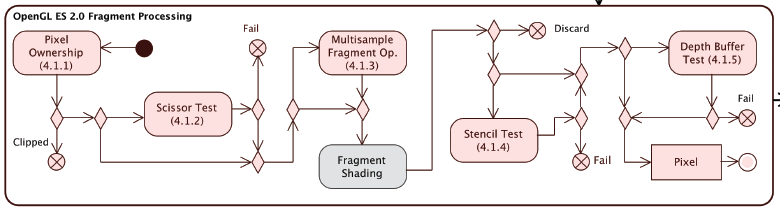


#### Primitive processing

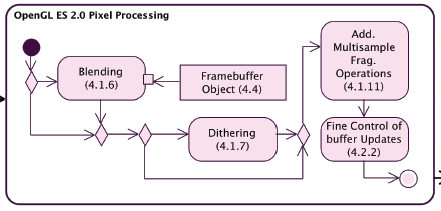


#### Rasterization



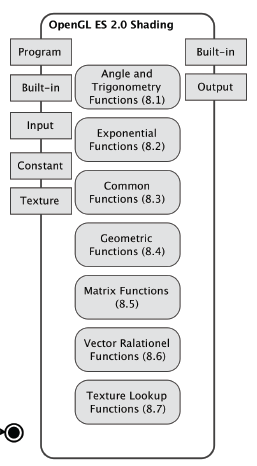
Fragment processing

#### Pixel processing



## OpenGL ES SL (GLSL ES)

The OpenGL ES Shading Language (GLSL ES) is based on OpenGL SL (GLSL) version 1.20. It is used to create shaders for vertex and fragment processors.

**

# Windows Systems/Libraries

One major benefit of OpenGL is the separation of the requirements of the *windowing system*, from the OpenGL *rendering model*. The windowing system is what is used to actually display graphics onto the screen and it handles user input. Within the windowing system the rendering model displays the pixels which OpenGL modifies. Every operating systems works libraries differently, to run programs on varying operating systems, the developer must use operation system specific functions. For example within, the Windows systems, applications are compiled using a library, essentially a .lib file. It also uses a driver at runtime a .dll for finding functions.

While each operating system uses different implementations of windowing systems. *GLUT* is cross platform library used displaying graphics in every operating system, unfortunately GLUT is not suitable for modern OpenGL, a newer version named *freeglut*.

### SFML

Whenever a new SMFL based window is opened, a context is automatically created. The primary drawback is that no control of the context is offered.

### SDL

SDL (Simple Direct Media Layer) , it has more control over the context, than SFML. It has several modules, when creating a context and window, the *video module* is used. It is supported by Windows, Linux and iOS devices. It is predominately used for video games.

### GLFW

As opposed to SFML and SDL only the basic requirements are offered.

* window creation
* context creation
* user input

The advantages of GLFW are that more control of the context is offered than the other libraries.

### GLUT

GLUT (Graphics Library Utility Toolkit) is a small toolkit, windowing system, for OpenGL applications. It creates a simple window, which is used for displaying OpenGL generated graphics to the screen. The main benefit of GLUT is that it is portable, so it can work on any device and operating system. The current version is 3.7.

With the modern systems both GLUT and freeglut use deprecated functions, today ***GLFW*** is now the most efficient window system it runs the all operating systems.

# Chapter 4: WebGL

## Introduction

WebGL (Web Graphics Library) is a very low level graphics API for displaying 2D and 3D graphics within the browser. WebGL accesses the graphics accelerator hardware from within the browser. WebGL is based on OpenGL ES 2.0, ES means *embed systems*, the browser can be interpreted as an embed system. Although it’s based on OpenGL ES 2.0 it has been modified to integrate with web browsers. Accessing the graphics accelerator from the browser allows WebGL to perform tasks typically allocated to other processes. OpenGL ES 2.0 exposes mobile devices (mobile phones, tablets, PDA’s) to displaying graphics.

The cross platform nature of WebGL allows it to be run on any Operating System, for example mobile phones, tablets and computers. It uses the GLSL ES shading language for realistic rendering. WebGL can only be rendered in the canvas element within the browser. It was created from canvas by Vlasimir Vukicevic while working at Mozilla. WebGL content is stored within the *rendering context* inside the HTML5 *Canvas* element. The WebGLRenderingContext interface features methods which allow graphics to be manipulated within the Canvas element.

|  |
| --- |
| **var** gl;  **try** {  gl = canvas.getContext("experimental-webgl");  } **catch**(e) {} |

## WebGL Viewport

Once the context is defined, the viewport is created a rectangle with origin at (0,0) and its height and width corresponds to *context.drawingBufferWidth, context.drawingBufferHeight*.

function initViewport(gl, canvas)

{

gl.viewport(0, 0, canvas.width, canvas.height);

}

## HTML5

Today modern webpages are scripted using HTML5 which is a new incarnation of HTML (Hyper Text Mark-up Language). HTML5 was created in 2008 it improves upon HTML in many aspects in regard to multimedia content by introducing elements for *audio* and *video* that defines both audio and video content in particular the *canvas* element which allows drawing within the browser using JavaScript. *“The most interesting was the inclusion of the <canvas> element. This element provides programmers with the ability for dynamic, scriptable rendering of 2D shapes and bitmap images.”* (Sunoallah, 2013)Additionally with the inclusion of WebGL to HTML5 a new type of array (Float32Array) was added to the HTML5.

## The Canvas element

The canvas element within HTML5 allows programmers to develop rich 2D graphics, gradients and animations and manipulate pixels, by using JavaScript. It is raster based with height and width properties. It is nothing more than an invisible block the default size is 300 X 150 pixels. To create graphics a *drawing context* must be defined. The below code defines a drawing context to create a simple circle.

requestAnimationFrame()

This function requests the browser to update the window for the next frame. The major benefit of this function is that all requests/call-backs are merged into one draw call which optimizing performance.

**The WebGL Context**

To implement OpenGL within the browser the canvas element is used. The method canvas.getContext() returns an object that allows for drawing to the canvas.

Canvas.getContext(contextType, contextAttributes);

Depending on the configuration of the browser the context will either be “experimental-webgl” or “webgl”. Experimental-webgl creates a WeGLRenderingContext object, this object is a three dimensional context, it is used for WebGL 1.0.

Additionally “experimental-webgl2” creates a WeGL2RenderingContext object, having three dimensions it is used for new experimental browsers for example Google Canary or Mozilla Firefox aurora. WebGL 2.0 (which is based on OpenGL ES 3.0)

## Shaders in WebGL

Shaders in WebGL are defined within the script tag, to denote the type of shader.

### Vertex shaders

<script id=”shader-vs” type=”x-shader/x-vertex”>

</script>

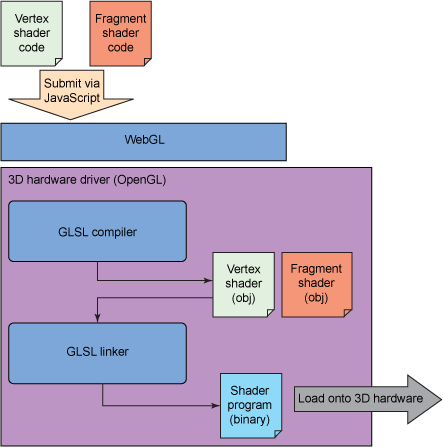
### Fragment Shaders

<script id=”shader-fs” type=”x-shader/x-fragment”>

</script>

### Execution model of shaders

Much the as in OpenGL except that it must pass through the browser via JavaScript



## Browser support

It currently is supported by all the leading desktop browsers. Browsers provide users with a “Black list and White List” which informs users of WebGL enabled devices. It is not supported by the following browsers

* Internet Explorer
* Desktop Safari on Windows
* Mobile Safari
* Opera for IOS
* Built-in Android browser

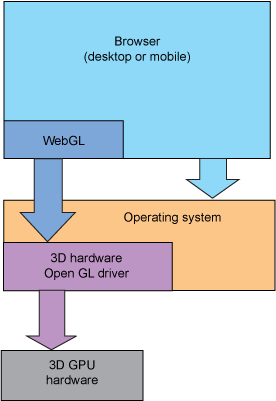
## Platform support

WebGL is a platform independent technology. It does work for the following

* Windows Phones
* IOS (IPhone, IPad)
* Windows 8 Modern UI

## WebGL Architecture

The web browser exposes the WebGL API via JavaScript. The WebGL API, consist of JavaScript functions, which are based on the OpenGL ES 2.0.



## Three.js

One major drawback of using WebGL is creating simple 3D environments can comprise of writing a hundred lines of code and the programmer is required to have an in-depth knowledge of computer graphics. Programming shaders, matrices and quaternions can be highly complex Three.js removes the strain of defining these objects. New frameworks developed for programmers with little computer graphics experience, allows for 3D environments to be easily created with a fraction of code compared to standard WebGL programs.

Three.js is a high level JavaScript library used for creating and displaying 3D graphics within the web browser. Three.js allows 3D graphics to be easily created without the assistance of other proprietary library and plugins it also has an open source licence. One of the primary benefits of Three.js is that it also features both a SVG, Canvas and WebGL renderers, where WebGL graphics must be within the canvas element of a HTML5.

### Three.js environment

Every Three.js environment is composed of the following.

#### Scene

Everything within this area will be rendered to the screen.

#### Camera

The typically represents a camera

The two types of cameras are perspective and orthographic.

#### Object

Every object in Three.js is comprised of geometry and a material. The geometry for example can be as simply cube or a sphere, three.js supports a wide variety of complex 3D objects.

## WebGL 2.0

Development commenced in 2013, it is currently under development, WebGL 2.0 is updated on a continuously basis. It will not be backwards compatible with WebGL 1.0.

# Chapter 5: Improving performance on the web

## Introduction

Due to the ever increasing advancements of web based technology which allows for faster processing power and now that JavaScript can be run on servers due to Node.js programmers are experimenting with creating interactive 3D experiences within the browser. Google Chrome is used primarily by developers of experimenting with 3D graphics within the browser, as it is slightly faster than Firefox. Almost all modern browsers support WebGL. Since 2014, Internet Explorer only supported 3D transforms, Internet Explorer version 11 supports WebGL.

Web based technologies continues to accelerate and adapt at ever increasing speeds to promote the performance of 3D graphics within the browser Mozilla and Epic Games Inc. worked together with the objective of porting native desktop games to the web browser. With the aim to create web based games that perform at speeds equal or at similar to native desktop games. *“This technology has reached a point where games users can jump into via a Web link are now almost indistinguishable from ones they might have had to wait to download and install”* (mozilla, 2014)

To do this they used the “Unreal Engine” (a program that creates video games and their components) developed by Epic Games and ported its technology to the web browser and included asm.js to achieve to great speed. Additionally Unity 5 (another program that creates video games) included asm.js into its platform to allow to games be run within the browser without any additional plugins. *“WebGL will become the prime solution for targeting the web with Unity.”* (ECHTERHOFF, 2014)

## Asm.js

Asm.js developed by Mozilla is a subset of JavaScript it was created to allow web apps to perform just as well as native desktop applications. Asm.js is implemented where high performance is essential for example browser based video games. *“Asm.js is a subset of JavaScript that is heavily restricted in what it can do and how it can operate. This is done so that the compiled Asm.js code can run as fast as possible making as few assumptions as it can, converting the Asm.js code directly into assembly.”* (Resig, 2013)

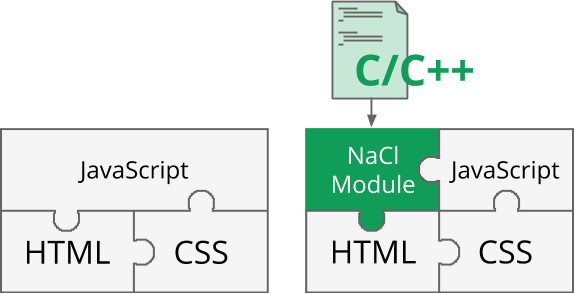
Asm.js is not a language that is not coded by programmers instead it is created by compilers, like Emscripten then executed within the browser. *“Optimizations have increased the performance of Web applications using asm.js from 40% to within 67% of native, and we expect it to get even faster.”* (mozilla, 2014)

## Emscripten

Emscripten developed by Mozilla is a complier which converts LLVM byte code into highly JavaScript. The LLVM bytecode is pre-complied C/C++ code into JavaScript. LLVM compiler byte code is finally converted into a highly optimized subset of JavaScript. *“Emscripten is a tool to convert C/C++ code to LLVM byte code and then to asm.js JavaScript code.”* (Mehrabani, 2014)

## Native Client (NaCI)

Google develop a technology allowing for C/C++ code to be run within a web browser via JavaScript referred to as Native Client or NaCI. Ultimately it allows for C/C++ scripted games to be accessed by a web browser without the requirement of a plugin. A typical NaCI application consists of HTML, CSS and JavaScript with an additional NaCI module. NaCI executes modules referred to as nexe, these module exist within a NaCI applications. Illustrated below.

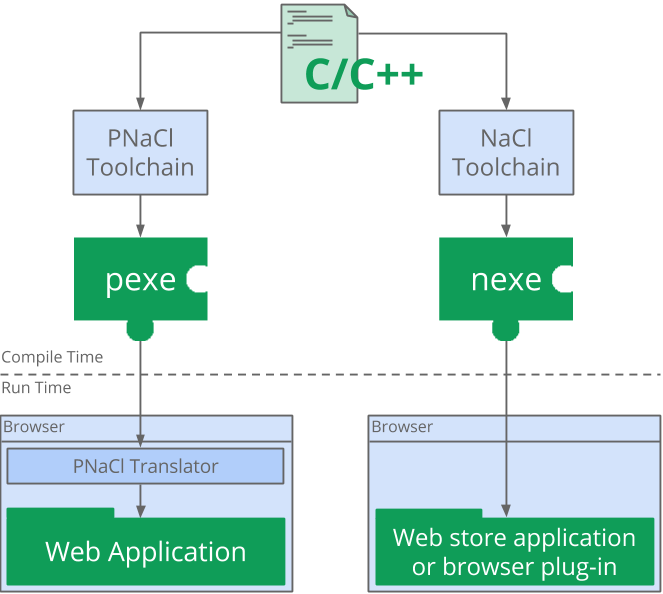


Left without NaCI and on the right with NaCI

There is also a portable version of NaCI referred to a Portable Native Client (PNaCL) (pronounced pinnacle) these applications can be added to any website. PNaCL allows for code to be complied only once which can then be run on any platform. PNaCL runs executable programs referred to as *pexe*. A compiler converts these pexe programs into code that the client hardware can interpret.

### How it works

This is a high level description of how NaCI operates. Within NaCI two components known as “*Toolchains”* and *“Runtime components”* work together to convert C/C++ applications to JavaScript based applications. Toolchains are a series of linkers, assemblers and compilers that convert C/C++ script to either Portable Native Client (PNaCl) or Native Client (NaCI) modules. Runtime components allows for these modules to be executed within the browser. *“Portable Native Client, as the name implies, solves the problem of how to deliver such code across multiple machine architectures. PNaCl compiles C or C++ code into what Google describes as "LLVM-style bytecode," an intermediate stage between the source code and machine code.”* (Yegulalp, 2013)



## Google ANGLE Project

The ANGLE (Almost Native Graphics Layer Engine) open source project developed by Google, translates WebGL and OpenGL 2.0 based API’s directly into DirectX 9 API. ANGLE is the default WebGL backend for Google Chrome and Mozilla Firefox on Windows based platforms. ANGLE is an implementation of OpenGL ES 2.0.

ANGLE was created because Windows based platforms could not run WebGL content, due to users not having the correct drivers installed. The main function of ANGLE is to allow Windows users to run WebGL content without having OpenGL drivers installed on their system. *“Even if they have powerful graphics hardware many Windows machines can’t render WebGL content because they don’t have the necessary OpenGL drivers installed… ANGLE should make it simpler to prototype these applications on Windows and also gives developers new options for deploying production versions of their code to the desktop.”* (Webb, 2010)

# Chapter 6: Methodology & Design

The aim of this paper is to investigate the differences in rendering performance between OpenGL and WebGL and to research how both API’s operate to render graphics on the based on a thorough analysis of theirs API’s specification. Research was undertaken into fundamentals of computer graphics and the rendering pipeline. Improving performance of the web pertains to how web technology is advancing to deliver web based applications that operate at optimal speeds and has features similar to their desktop applications counterparts.

# Research Question

The primary goal of this research paper is to analyse the performance of 3D graphics both within the web browser and desktop environment, and monitor the GPU and CPU performance. Additionally to analyse how both API’s accesses the GPU to display graphics to the screen. To investigate how WebGL operates on different browsers and also to investigate how both API’s operates on different operating systems.

The metrics to be analysed will be based on the performance of the GPU and CPU needed to render graphics. These metrics will be gathered by tracing the method calls and documenting the processing time of these calls. The expected outcome is that OpenGL will be far more effective at rendering graphic, due to the fact that WebGL additionally needs to pass through the web browser.

## The Approach

The proposed approach to determine the difference in performance is to create a series of varying programs that function effectively until the optimal frame rate decrements. For example one program could indefinitely add objects to a scene the addition of more objects to a scene is directly proportional to the performance strain on the computer which could be analysed. The function of these programs will pertain to distinct operations of the rendering pipeline and other areas of OpenGL and WebGL. The primary metrics used to analyse performance is the framerate and CPU and GPU performance.

# Design

### Risk Analysis

There is an array of risks in the implementation and analysis with this project.

* Testing with outdated computers with outdated drivers.
* Developing and testing with a laptop that has a slower GPU than current laptops.

### UML Diagrams

(Within technical document)

### User Stories

(Within technical document)

### Test Plans

(Within technical document)

### VOPCs/ Class Diagram

(Within technical document)

# Chapter 7: Implementation

## Sprints/Phases

GitHub was used for source control all commits/updates are held within the development folder. Sprints are broken down into distinct phases these phases are categorized into phase one, two three.

When implementing the OpenGL project the **freeglut** library and driver was included in order to display graphics to the screen. Freeglut is a revision of the **glut** library.

When developing the WebGL projects the “**webgl-utils.js**” file was included which provides consistent functionality of the **RequestAnimationFrame** method on varied web browsers. For debugging purposes the “**webgl-debug.js**” was included. For matrix calculations the **“glMatrix-0.9.5.min.js” whi**

Sp

# Chapter 8 : Analysis and Findings

# Benchmarks

The measurement of performance of an application is referred to as benchmarking. One example of benchmarking is measuring how fast primitives can be rendered to the screen and measuring the frame rate of applications.

Matrix calculation speeds

Amount of draw calls

Processing time

Google V8 engine vs C++ compiler

# Metrics

CPU performance

GPU performance

Vertex and fragment shader compilation speeds

Fragment shader utilization

Frames per seconds

Number of OpenGL function calls per frame

# References

Apple Inc. (2012). OpenGL Programming Guide for Mac. 182.

Conrad, D. (2014, June 4). *Getting started with WebGL*. Retrieved from http://www.i-programmer.info/: http://www.i-programmer.info/programming/graphics-and-imaging/1970-getting-started-with-webgl.html?start=1

*Definition of:graphics pipeline*. (2001). Retrieved from http://www.pcmag.com/: http://www.pcmag.com/encyclopedia/term/43933/graphics-pipeline

ECHTERHOFF, J. (2014, April 29). *ON THE FUTURE OF WEB PUBLISHING IN UNITY.* Retrieved from http://blogs.unity3d.com/: http://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/

Hock-Chuan, C. (2012, July). *3D Graphics with OpenGL*. Retrieved from http://www.ntu.edu.sg/: http://www.ntu.edu.sg/home/ehchua/programming/opengl/cg\_basicstheory.html

Jakober, L. (2011, March 9). *Can Web applications rival the performance of native and desktop applications?* Retrieved from http://apmblog.compuware.com/: http://apmblog.compuware.com/2011/03/09/can-web-applications-rival-the-performance-of-native-and-desktop-applications/

Jørgensen, M. N. (2012). *webgl based 3D game engine.*

Mallet, I. (2014). *Rasterization Stage:*. Retrieved from http://www.geometrian.com/.

Mehrabani, A. (2014, May 16). *Understanding asm.js*. Retrieved from sitepoint: http://www.sitepoint.com/understanding-asm-js/

mozilla. (2014, March 12). *Mozilla and Epic Preview Unreal Engine 4 Running in Firefox.* Retrieved from blog.mozilla.org: https://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-firefox/

Resig, J. (2013, April 3). *Asm.js: The JavaScript Compile Target.* Retrieved from http://ejohn.org/: http://ejohn.org/blog/asmjs-javascript-compile-target/

Richard S. Wright, J., Haemel, N., Sellers , G., & Lipchak, B. (2013). *OpenGL SUPERBIBLE Fifth Edition.*

*Section 1.1 OpenGL History*. (2010, January 20). Retrieved from www.yaldex.com: http://www.yaldex.com/open-gl/ch01lev1sec1.html

Segal, M., & Akeley, K. (2004). *The OpenGL Graphics System: A Specification Version 2.0 - October 22, 2004.*

Sloup, P. (2011). *WebGL Earth.*

Sunoallah, S. J. (2013). *Design and Implementation of a Standard Library for WebCube Components and Events.* Gaza.

Tchernitski, V. (2013, February 17). *Hydrodynamic Process Simulation on a Mobile Device*. Retrieved from http://rnd.azoft.com/fluid-dynamics-simulation-on-ios/

Webb, M. (2010, March 30). *Google goes 3D with launch of ANGLE graphics driver project*. Retrieved from http://www.gizmag.com/: http://www.gizmag.com/google-chrome-angle-project-webgl/14665/

Yegulalp, S. (2013, Nov 13). *Chrome 31 pushes Web apps a step closer to desktop apps*. Retrieved from http://www.infoworld.com/article/2609713/web-browsers/chrome-31-pushes-web-apps-a-step-closer-to-desktop-apps.html

# Bibliography

Apple Inc. (2012). OpenGL Programming Guide for Mac. 182.

Conrad, D. (2014, June 4). *Getting started with WebGL*. Retrieved from http://www.i-programmer.info/: http://www.i-programmer.info/programming/graphics-and-imaging/1970-getting-started-with-webgl.html?start=1

*Definition of:graphics pipeline*. (2001). Retrieved from http://www.pcmag.com/: http://www.pcmag.com/encyclopedia/term/43933/graphics-pipeline

ECHTERHOFF, J. (2014, April 29). *ON THE FUTURE OF WEB PUBLISHING IN UNITY.* Retrieved from http://blogs.unity3d.com/: http://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/

Hock-Chuan, C. (2012, July). *3D Graphics with OpenGL*. Retrieved from http://www.ntu.edu.sg/: http://www.ntu.edu.sg/home/ehchua/programming/opengl/cg\_basicstheory.html

Jakober, L. (2011, March 9). *Can Web applications rival the performance of native and desktop applications?* Retrieved from http://apmblog.compuware.com/: http://apmblog.compuware.com/2011/03/09/can-web-applications-rival-the-performance-of-native-and-desktop-applications/

Jørgensen, M. N. (2012). *webgl based 3D game engine.*

Mallet, I. (2014). *Rasterization Stage:*. Retrieved from http://www.geometrian.com/.

Mehrabani, A. (2014, May 16). *Understanding asm.js*. Retrieved from sitepoint: http://www.sitepoint.com/understanding-asm-js/

mozilla. (2014, March 12). *Mozilla and Epic Preview Unreal Engine 4 Running in Firefox.* Retrieved from blog.mozilla.org: https://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-firefox/

Resig, J. (2013, April 3). *Asm.js: The JavaScript Compile Target.* Retrieved from http://ejohn.org/: http://ejohn.org/blog/asmjs-javascript-compile-target/

Richard S. Wright, J., Haemel, N., Sellers , G., & Lipchak, B. (2013). *OpenGL SUPERBIBLE Fifth Edition.*

*Section 1.1 OpenGL History*. (2010, January 20). Retrieved from www.yaldex.com: http://www.yaldex.com/open-gl/ch01lev1sec1.html

Segal, M., & Akeley, K. (2004). *The OpenGL Graphics System: A Specification Version 2.0 - October 22, 2004.*

Sloup, P. (2011). *WebGL Earth.*

Sunoallah, S. J. (2013). *Design and Implementation of a Standard Library for WebCube Components and Events.* Gaza.

Tchernitski, V. (2013, February 17). *Hydrodynamic Process Simulation on a Mobile Device*. Retrieved from http://rnd.azoft.com/fluid-dynamics-simulation-on-ios/

Webb, M. (2010, March 30). *Google goes 3D with launch of ANGLE graphics driver project*. Retrieved from http://www.gizmag.com/: http://www.gizmag.com/google-chrome-angle-project-webgl/14665/

Yegulalp, S. (2013, Nov 13). *Chrome 31 pushes Web apps a step closer to desktop apps*. Retrieved from http://www.infoworld.com/article/2609713/web-browsers/chrome-31-pushes-web-apps-a-step-closer-to-desktop-apps.html