

Experimental Approach of the Asymptotic Computational Complexity of Shaders for Mobile Devices with OpenGL ES

Alex S. Campelo
University of Brasilia

Edson A. Costa
University of Brasilia

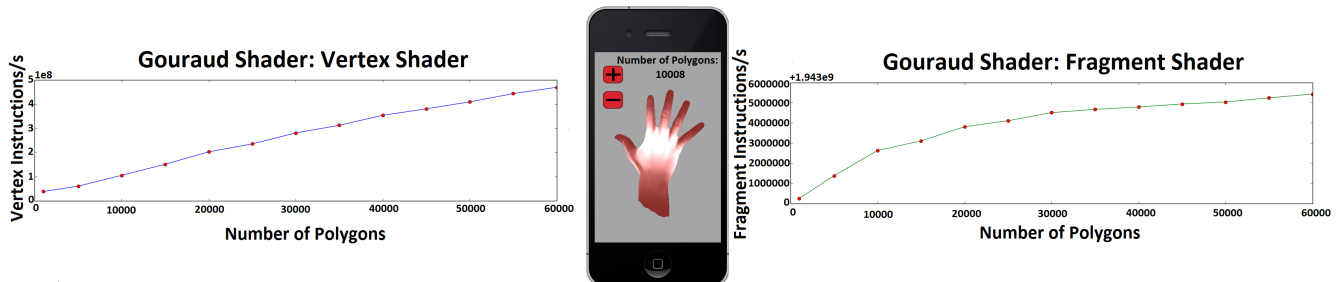


Figure 1: *Gouraud shading analysis*

Abstract

The usage of mobile devices and increasingly realistic graphics is emerging, but the graphics performance is still a critical factor in games. There's more hardware restriction on mobile devices than on a computer. Thus, this paper proposes an experimental approximation of the asymptotic computational complexity of miscellaneous vertex and fragment shaders for Android and iOS platforms. The asymptotic complexities of the shaders will be analyzed based on number of instructions per second and rendering time metrics, depending on the number of polygons rendered. By means of the adjusted curves is also possible to compare the performance of the devices used in this work, which are the Nexus 4, HTC One, iPhone 5s and iPad Air. Besides, an automatic tool – that plots the data and uses the method of least squares to adjust the values obtained – will be presented, being able to estimate which curve has better approximation to the sampled data.

Keywords: Android, iOS, shaders, mobile devices, computer graphics, games, asymptotic complexity.

Author's Contact:

{campelo.al1,prof.edson.alves.costa}@gmail.com

1 Introduction

Graphics in games are so important that can determine the game's failure or success [Sherrod 2011]. Thus, the creation of three-dimensional scenes, using mobile devices, is becoming more and more usual and realistic [Sinthanayothin et al. 2012]. However, there're hardware restrictions, especially in mobile devices. Rendering graphics for mobile devices is still a challenge due to limitations, when compared to a computer, related to CPU (Central Processing Unit), GPU (Graphics Processor Unit) and power consumption [Arnaud et al. 2013].

In this context, graphics performance is a key factor for the overall performance of the system, mainly in games, which also has other factors that consume resources, such as artificial intelligence, networking, audio, input events, physics, among others.

But the recent growing of mobile devices made them able to support applications even more complex. Devices like smartphones and tablet computers have been widely adopted, emerging as one of the most propagated technologies. Within this context, the most commonly used mobile operating systems are iOS and Android platforms. Accordingly Apple's CEO, Tim Cook, more than 800 million iOS device were already sold¹ and the daily activation de-

vices using Android platform is approximately 1.5 million [Sandberg and Rollins 2013].

This way, it's possible to analyze the performance of the rendering process done by the GPU – in which different shaders (responsible for the creation of visual effects) are applied. Then, the goal of this work is to analyze the asymptotic computational complexity of shaders for mobile devices, both for the whole rendering process and for only part of it (vertex and fragment shaders).

2 Related Work

As said before, the game industry have sought to create games with a high level of realism. One of the factors that contributes to the increase of this realism was the introduction to programmable hardware, that allowed to program the rendering process. This way, the visual effects were the focus in [N. and S. 2007], where were presented diverse realistic and non-realistic techniques used in games in the last years. To achieve those effects, the programmable rendering pipeline were used, specifically the vertex and fragment shaders.

[Com 1999] shows an approach to measure graphics performance, which says to develop a program that makes graphics calls and measure the performance of the system running this program. If industry standard graphics benchmarks are used, performance of many different systems can be compared with minimal effort. The graphics hardware performance is measured in terms of maximum rate the system can achieve in drawing, like vectors/second, shaded triangles/second, by example.

The shader performance analysis were already done in [N. et al. 2011], but related to the Tessellator shader. This shader is available on OpenGL 4 and allows the creations of vertex directly on the GPU, reducing the amount of transfer between CPU-GPU. The performance was analyzed increasing the number of three-dimensional objects, in other words, the number of polygons. The chosen metric to analyze the performance was the frames per second.

3 Background Information

This section gives some brief background information that is needed to understand certain parts of the work. It describes the definition of shader, asymptotic complexity and the least squares method.

3.1 Shader and OpenGL ES

Shading is the process of using an equation for computing the surface behavior of an object [Moller et al. 2008]. Shader algorithms

update

¹<http://www.theverge.com/2014/6/2/5772344/apple-wwdc-2014-stats>

are written by the programmer to override the predefined functionality of the rendering process performed by the GPU, by the usage of graphic libraries such as OpenGL ES.

Before the shaders were created, the rendering pipeline was completely fixed. But with the introduction of the shaders, it's possible to customize part of this process, like the vertex and fragment processing (vertex and fragment shaders).

The OpenGL ES (OpenGL for Embedded Systems) were released in 2003, being the OpenGL version to embedded systems. As said by [Guha 2011], the OpenGL ES is one of most popular API (Application Programming Interface) for graphics programming in mobile devices. It uses the GLSL (OpenGL Shading Language) as shading language, that is based on the C language.

3.2 Asymptotic Complexity

Asymptotic complexity is a way to compare the efficiency of an algorithm, in terms of time, memory or processing, by example. To not depend on the platform nor programming language, the asymptotic complexity is based on a function (logic measure) [Drozdek 2002]. It expresses a relationship between the amount of data and time required to process them.

The calculation of the asymptotic complexity aims to model the behaviour of the algorithm performance, as the number of data increases. This way, the terms that doesn't affect the order of magnitude are eliminated, generating the approximation called asymptotic complexity. The Equation 1

$$y = n^2 + 10n + 1000 \quad (1)$$

could be approximated by the Equation 2.

$$y \approx n^2 \quad (2)$$

3.3 Least Squares Method

The least squares method is used to adjust a set of points (x, y) to a determined curve. In linear adjustment case, by example, represented by $y = a + bx$, many points are not collinear. As said in [Rorres 2001], it's impossible to find coefficients a and b that satisfy the system. Thus, the distances between those values to the line can be considered as error measures and the points are adjusted by the same vector. This way, there's a linear least squares adjustment to the data and its solution is given in Equation 3.

$$v = (M^T M)^{-1} M^T y \quad (3)$$

where

$$M = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, \quad v = \begin{bmatrix} a \\ b \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad (4)$$

The adjustment to a second and third degree functions is similar, but the M matrix is redefined to Equation 5 and Equation 6, respectively.

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix} \quad (5)$$

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 \end{bmatrix} \quad (6)$$

The exponential adjustment is a little bit different. [Leithold 1994] says that the exponential function can be represented as the Equation 7, where e , c , k are constants. Applying the logarithmic function on both sides, the Equation 8 is obtained and it can be simplified as the Equation 9 (where \bar{b} is a new constant). This equation is equivalent to a linear equation and the least squares method can be applied.

$$y = ce^{-kt} \quad (7)$$

$$\ln y = \ln c + \ln e^{-kt} \quad (8)$$

$$\bar{y} = \bar{a} + \bar{b}t \quad (9)$$

The final values of the \bar{a} and \bar{b} coefficients determine the c and k parameters through these relationships:

$$c = e^{\bar{a}} \quad e^{-\bar{b}} = -k \quad (10)$$

4 Development

This section describes the steps taken in this work, showing since the equipment used until the implementation, collection and analysis of data.

4.1 Equipment Used

The computer used for development on Android platform was an Alienware M14x manufactured by Dell with Intel Core i7 processor, 2GB of GeForce GTX as GPU and 8 GB of RAM. For the development on iOS platform, a Macbook Pro 11.1 was used, with Intel Core i5 processor and 8GB of RAM.

The Table 1 shows the used devices, which are equipments with different resolutions and hardware configurations. The benchmark app called 3D Mark was used to compared the different performances of these devices. It runs several graphical tests, in order to stress the GPU and to give a final punctuation related to the performance. The higher the score, the better the performance. This score is shown in the Table 2

Table 1: Mobile devices

Device	Platform	Resolution	GPU
Nexus 4	Android	768 x 1280	Adreno 320
HTC One	Android	1080 x 1920	Adreno 320
iPad Air	iOS	2048 x 1536	PowerVR G6430
iPhone 5s	iOS	1136 x 640	PowerVR G6430

Table 2: Benchmark

Device	Score
Nexus 4	7.106
HTC One	10.184
iPad Air	14.952
iPhone 5s	14.750

4.2 Android Implementation

To make the asymptotic computational complexity analysis possible, firstly was necessary to implement the shaders on Android platform. This was done using the graphics library called OpenGL ES. The object-oriented paradigm was used and the Figure 2 shows the class diagram and how the code was structured. This diagram presents a set of classes and their relationships, being the central diagram of object-oriented modeling.

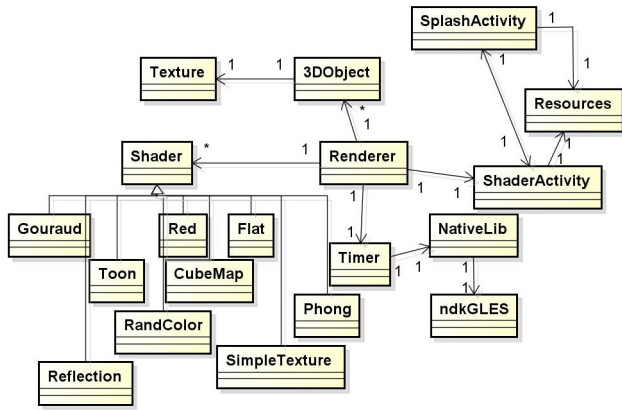


Figure 2: Android implementation: class diagram

4.2.1 Front-end Screen

The front-end screen is responsible for the interaction with the user, passing the input information to the back-end. The Android platform uses the term Activity to describe the application's front-end screen. It has design elements like text, buttons, graphics, among others. In this work, there're two Activity classes: Shader Activity and Splash Activity (Figure 3).

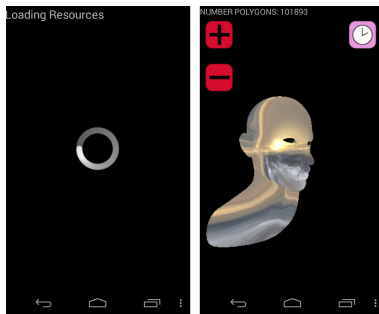


Figure 3: Shader Activity and Splash Activity

The Splash Activity is responsible for the visualization of the loading screen while the necessary resources – like the three-dimensional objects and textures – are loaded by a thread. This resources are managed by the Resources class, that uses the project pattern called Singleton. This pattern ensures that there's only one instance class, which will be accessed later.

The Shader Activity is responsible for creating an instance of the Renderer class, which renders the three-dimensional objects. Besides, it controls the touch events, that allows scaling and rotating these objects. It also shows the buttons that increase and decrease the number of polygons.

4.2.2 Three-dimensional Object

The three-dimensional object is represented by the composition of the 3DObject and Texture classes. The 3DObject class is responsible for reading and interpreting the obj files, that contains the information about the object. After this, the position, normal and texture vertices are stored into a buffer. The Texture class generates the textures, used by some shaders, from images. Those images are created for each three-dimensional model, using the UV Mapping technique. It maps the texture coordinates to an image, like is shown on Figure 4.

4.2.3 Renderer

The Renderer class works like a controller, being responsible for the rendering. It is the main class for the calls from the view (Shader Activity) and model (3DObject, Shader and Timer) classes. This class implements the functions from the OpenGL

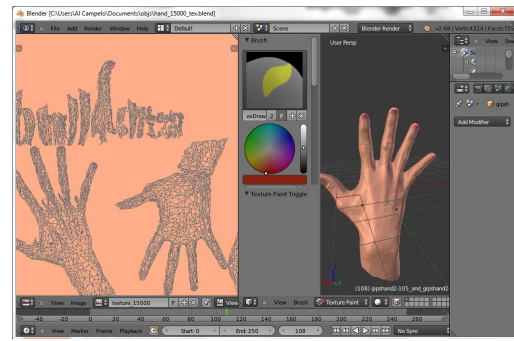


Figure 4: UV Mapping technique

ES library onSurfaceCreated(), onDrawFrame() and onSurfaceChanged().

4.2.4 Shader

The Shader class reads, attaches and links the vertex and fragment shaders. Furthermore, it has the abstract methods getParamsLocation() and initShaderParams(Hashtable params). The first method stores the location of each variabel specified in the shader. The second method initializes these variables based on a hash, which contains the values for each variable. This way, every shader inherits from the Shader class and must implement these abstract methods. The implemented shaders can be seen on Figure 5.

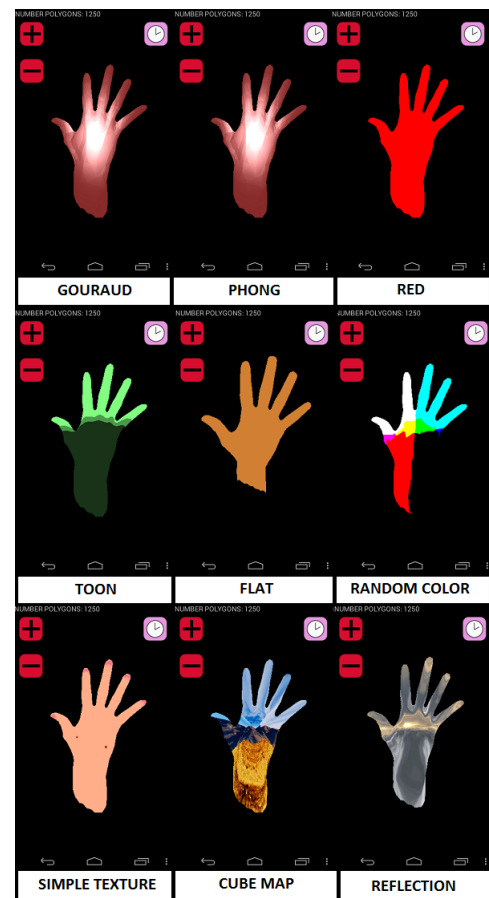


Figure 5: Implemented Shaders

4.2.5 Calculation of Rendering Time

The Timer class measures the rendering time in nanoseconds. Each measurement is done using the C language and the OpenGL ES extension called GL_EXT_disjoint_timer_query. The integration between the code in C language and the code in Java is done

by the class called NativeLib. If the extension is not available for the device, an alert is issued.

4.3 iOS Implementation

The structure of the code on iOS platform is similar to the Android, as is shown in Figure 6. It follows the Model-View-Controller pattern, which the controller is responsible for the integration between the Shader, 3DObject classes and the view RendererView.

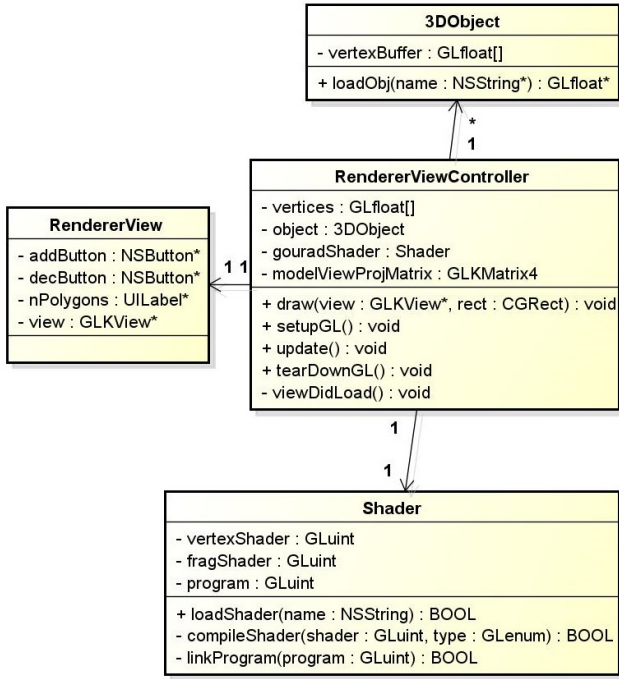


Figure 6: iOS implementation: class diagram

The 3DObject class interprets the obj file to the format accepted by OpenGL ES. The Shader class, as in Android platform, reads, attaches and links the vertex and fragment shaders.

The Gouraud Shader was chosen to be implemented and posteriorly to do the comparisons between the different devices on distinct platforms. The result is seen on Figure 7

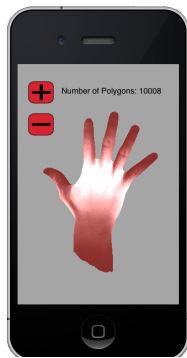


Figure 7: Gouraud Shader on iOS platform

4.4 Experimental Estimation of Asymptotic Complexity

The experimental estimation of asymptotic complexity of each shader was done by diverse measurements for each polygon counting (represented by each three-dimensional model). The asymptotic complexity was analyzed by two points of view: related to the entire rendering process and only related to the vertex and fragment shaders.

4.4.1 Rendering Process

In Android platform, as mentioned in Section 4.2.5, an OpenGL ES extension was used to get the rendering process time, done by the `glDrawArrays()` function. In iOS platform, the module – from Xcode development tool – called Instruments was used, which informs the elapsed time of each OpenGL ES function in microseconds.

This way, the measures were gathered for the devices Nexus 4, iPhone 5s and iPad Air. It wasn't possible to collect for the HTC One, because the extension wasn't available for this Android device.

4.5 Vertex and Fragment Shaders

The vertex and fragment shaders measurements were only possible to do in Android devices, because the Instruments module of Xcode doesn't exhibit any information about them. Then, the tool used to collect the measures was the Adreno Profile, because the GPU of the devices used is Adreno.

The chosen metrics were instructions per second per vertex and instruction per second per fragment. These metrics were gathered for each polygon counting, being exported in CSV (Comma-Separated Values) format.

4.6 Plot

After the measurements were done, the charts were plotted both for the rendering process, as for the vertex and fragment shaders. The first set of charts is related to the time, in nanoseconds, versus the polygon count. The second is related to the number of instructions per second per vertex (or fragment) versus the polygon count.

4.7 Automation of Curves Adjustments

To do the curve adjustment, was used the least squares method to linear, quadratic, cubic and exponential functions. The squared errors associated to each adjustment were also calculated, in order to determine which function had a better approximation to the original measures. Smaller the error, the better the approximation.

A program in Python was created to automate this calculation process. It reads CSV or txt files, calculates the average of the measures, plots the charts and does the curve adjustment (also plotting the data). The program is command-line based, having as parameters, the shader name and the measurement used (if it's related to the whole rendering process or just to the vertex and fragment shaders).

The Code 1 shows two command-lines examples, the first is related to the rendering process of the Gouraud shader and the second is related only to the vertex and fragment shaders.

Code 1: Command-lines

```

$ python shaderComplexity.py gouraud
$ render_time
$ python shaderComplexity.py gouraud
$ vertex_fragment
  
```

The Figure 8 shows how this program is structured. The ReadCSV and ReadTxt classes are responsible for reading the CSV and .txt files. The PlotChart class plots the original and adjusted datas and the LeastSquares calculates these adjustments and their errors.

In Figure 9 the result of the tool for the rendering process is presented, which are four screens. The first one (topdown) is the plot of the linear adjustment, the second is the cubic adjustment, the third is the quadratic adjustment and the last one is the exponential adjustment.

In Figure 10 and Figure 11 the results of the tool for the vertex and fragment shaders are presented, which are four screens as well. The first one (topdown) is the plot of the linear adjustment, the second is the quadratic adjustment, the third is the cubic adjustment and the last one is the exponential adjustment.

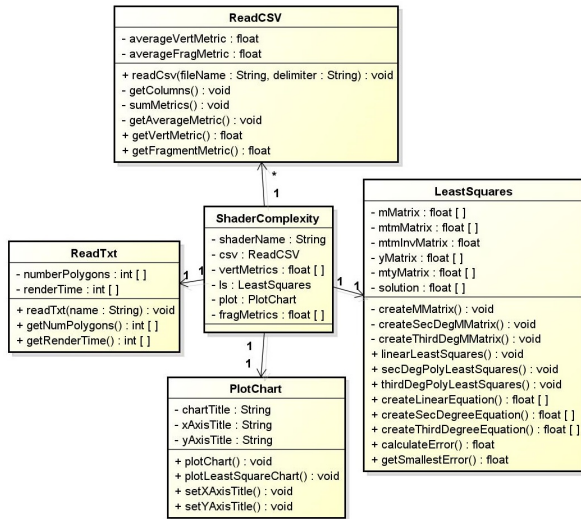


Figure 8: Tool implementation: class diagram

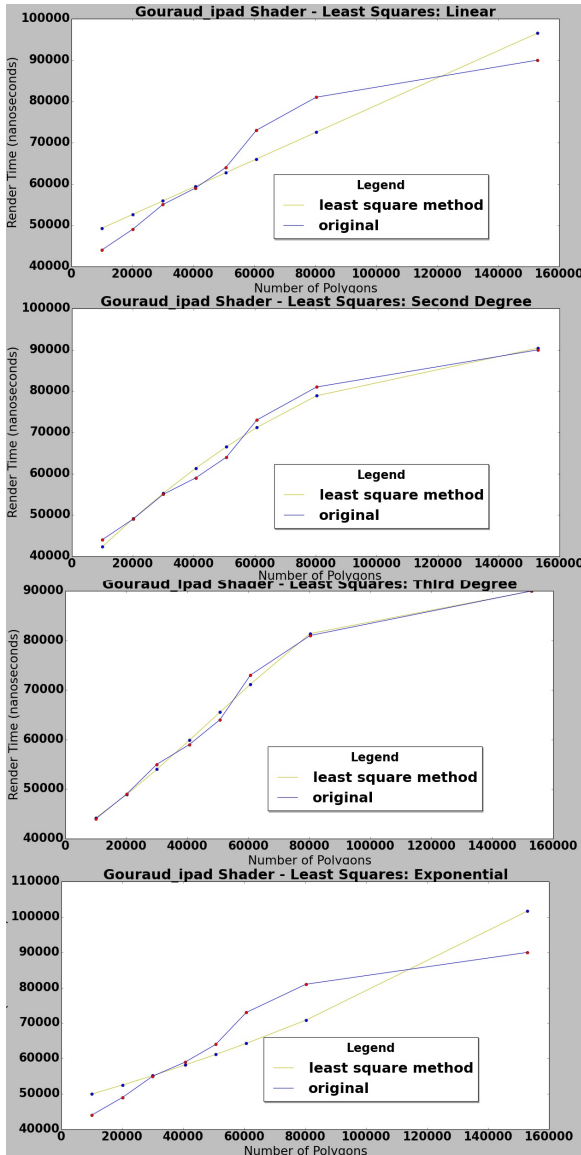


Figure 9: Automatic Adjustments: rendering process

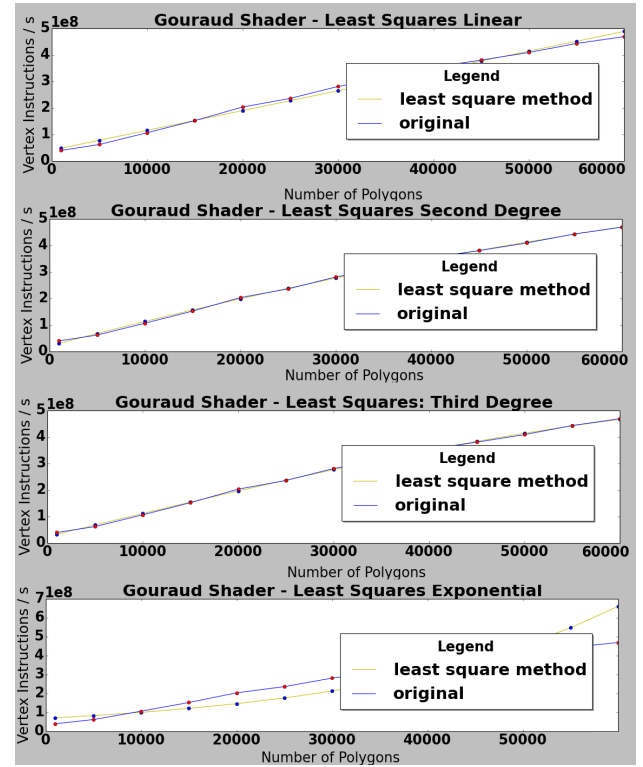


Figure 10: Automatic Adjustments: vertex shader

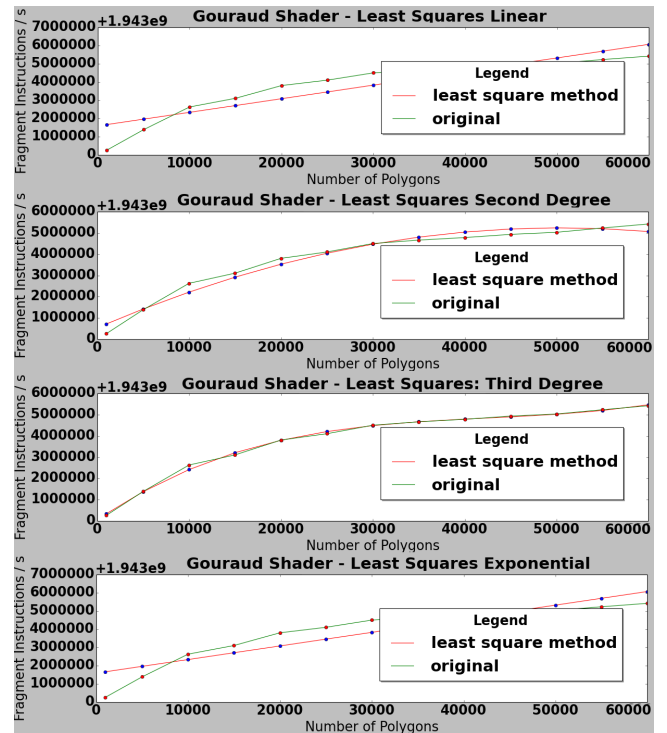


Figure 11: Automatic Adjustments: fragment shader

5 Results

For each shader were plotted the charts related to the entire rendering process and to the vertex and fragment shaders for different devices. After these plots, it was noticed that the charts for all shaders and devices had similar curves for each measure type (rendering process, vertex and fragment shader).

5.1 Android Devices

With the Nexus 4 device was possible to plot the charts related to the rendering process and to the vertex and fragment shaders. The charts about the vertex shader visually resulted in a linear function (with different slopes). The Figure 12 shows the charts related to the vertex shader of all implemented shaders.

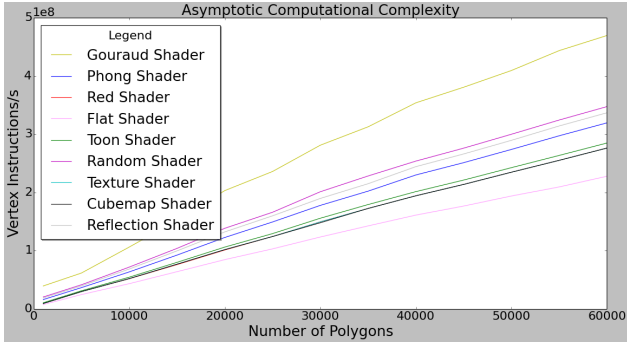


Figure 12: Vertex shader: curves comparison

The curves related to the rendering process and to the fragment shader had similar shapes, but it wasn't possible to determine the exact curve only by visual inspection. These curves are shown in the Figure 13 and Figure 14.

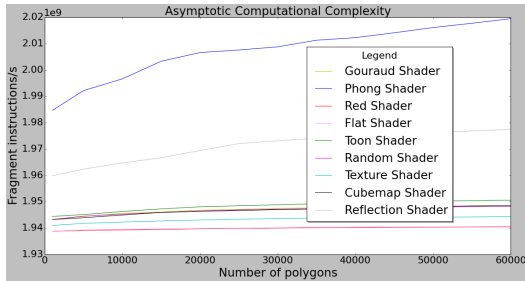


Figure 13: Fragment shader: curves comparison

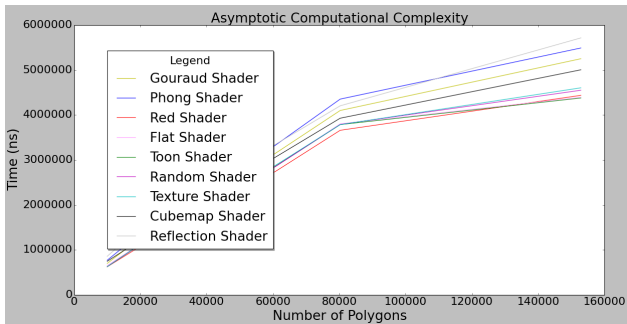


Figure 14: Rendering process: curves comparison

Then, the adjustments to the predefined curves were done by the automated tool and plotted for each shader. The smallest errors were also determined, in order to discover which curve had the best approximation. By this analysis, all the shaders had better approximation to a third degree curve, both for the fragment shader as for the rendering process.

For the HTC One device was only possible to measure the performance related to the vertex and fragment shader. The results

were the same as in the Nexus 4, which the vertex shader had a linear behavior and the fragment shader, a cubic behavior (Figure 15).

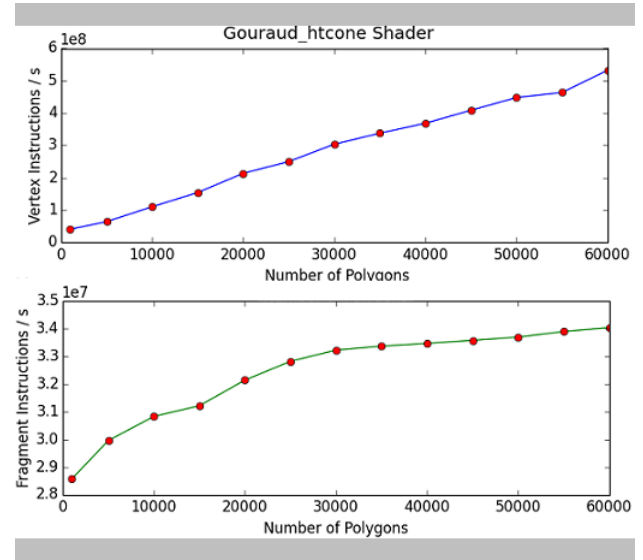


Figure 15: HTC One device

5.2 iOS Devices

With the iOS devices, it was only possible to plot the charts related to the rendering process. The shapes of the obtained curves are similar to the obtained curves in Nexus 4, and the best approximations were also to a third degree curve. The Figure 16 shows these curves for the iPhone 5s and for the iPad Air.

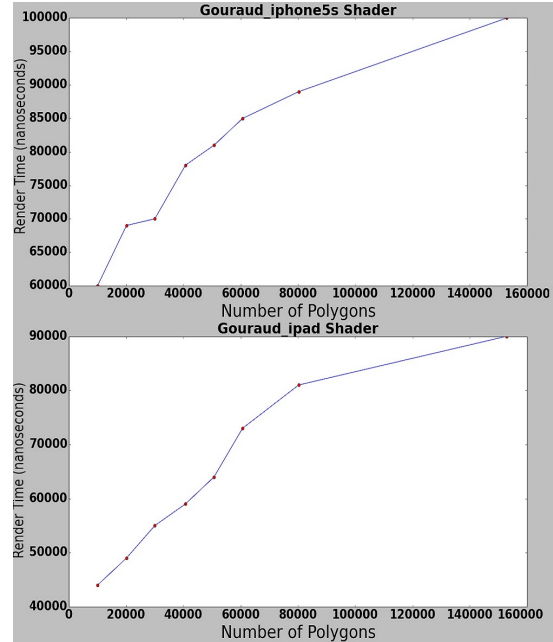


Figure 16: iOS devices

5.3 Analysis of the Equations

With the automated tool, it was also possible to calculate the equations of the adjusted curves, for each shader of the Nexus 4. They are shown in Table 3, Table 4 and Table 5. Although the curves are of the same family, their coefficients are not identical. The shaders relatively more complexes had steeper slopes compared to the simple shaders and had greater x^2 and x^3 terms.

Analyzing the equations, it's possible to see that the vertex shader that had better performance was the Flat Shader, which only

Table 3: Equations related to the vertex shader

Shader Name	Vertex Instructions per Second
Gouraud	$y = 40.16 \times 10^6 + 7486.43n$
Phong	$y = 14.95 \times 10^6 + 5211.02n$
Red	$y = 8.02 \times 10^6 + 4545.69n$
Toon	$y = 10.17 \times 10^6 + 4673.96n$
Flat	$y = 7.65 \times 10^6 + 3738.61n$
Random Color	$y = 20.58 \times 10^6 + 5640.13n$
Simple Texture	$y = 8.80 \times 10^6 + 4540.32n$
CubeMap	$y = 8.67 \times 10^6 + 4540.40n$
Reflection	$y = 18.03 \times 10^6 + 5470.95n$

Table 4: Equations related to the vertex shader

Shader Name	Fragment Instructions per Second
<i>Gouraud</i>	$y = 19.43 \times 10^8 + 297.00n$ $-0.0065n^2 + 0.50 \times 10^{-7}n^3$
<i>Phong</i>	$y = 19.84 \times 10^8 + 1752.43n$ $-0.0389n^2 + 3.32 \times 10^{-7}n^3$
<i>Red</i>	$y = 19.39 \times 10^8 + 64.34n$ $-0.00090n^2 + 0.05 \times 10^{-7}n^3$
<i>Toon</i>	$y = 19.44 \times 10^8 + 268.89n$ $-0.0044n^2 + 0.30 \times 10^{-7}n^3$
<i>Flat</i>	$y = 19.39 \times 10^8 + 74.94n$ $-0.0013n^2 + 0.08 \times 10^{-7}n^3$
<i>Random Color</i>	$y = 19.43 \times 10^8 + 250.33n$ $-0.0050n^2 + 0.37 \times 10^{-7}n^3$
<i>Simple Texture</i>	$y = 19.41 \times 10^8 + 160.00n$ $-0.0030n^2 + 0.22 \times 10^{-7}n^3$
<i>CubeMap</i>	$y = 19.43 \times 10^8 + 245.89n$ $-0.0047n^2 + 0.37 \times 10^{-7}n^3$
<i>Reflection</i>	$y = 19.59 \times 10^8 + 698.57n$ $-0.0094n^2 + 0.47 \times 10^{-7}n^3$

determines the x and y coordinates (since the z is zero). The vertex shader that had worst performance was the Gouraud Shader, which calculates the light components in the vertex shader.

The shader with better performance – related to the fragment shader – was the Red Shader, in which only determines the fragment color to red. The fragment shader with worst performance was the Phong Shader, which does the same calculation as the Gouraud shading but in the fragment shader instead of the vertex shader.

The shaders with better performance – related to the rendering process – were the Flat, Toon and Red Shaders. The shaders with worst performance were the Reflection and Gouraud Shaders.

Besides, with the equations, it's possible to estimate the number of vertex or fragment instructions per second. Taking the Toon Shader as example, which its vertex shader equation is $y(n) = 10.17 \times 10^6 + 4673.96n$, the estimated number of instructions per second is for 60000 polygons is 29.06×10^7 . With the tool Adreno Profiler it was possible to see that this value is close to the measured (28.49×10^7).

Another relevant result is about the Gouraud and Phong Shaders. The first had the worst vertex shader performance and the second had the worst fragment shader performance. But the shader that had the worst rendering process performance was the Phong Shader. This result is consistent because the fragment shader, by this experiment, has computational complexity $O(n^3)$ and the vertex shader, $O(n)$, influencing this worst outcome.

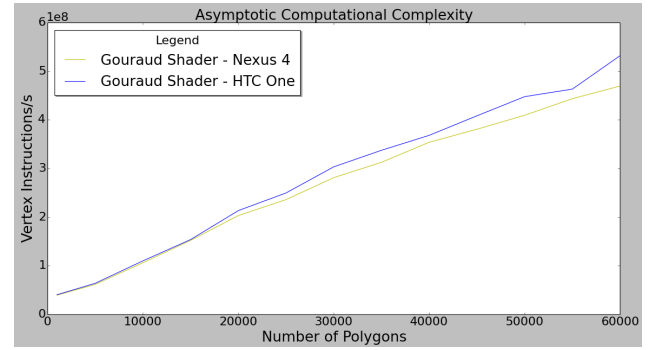
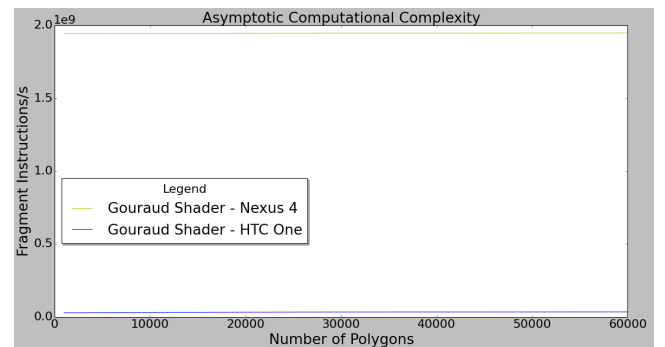
5.3.1 Devices Comparison

With the obtained curves and equations, it was also possible to compare the devices. The Figure 17 and Figure 18 shows the curves related to the vertex and fragment shaders for the Nexus 4 and HTC

Table 5: Equations related to the rendering process

Shader Name	Rendering Process Time (ns)
Gouraud	$y = 24.31 \times 10^4 + 48.89n + 7,60 \times 10^{-5}n^2 - 1.19 \times 10^{-9}n^3$
Phong	$y = 31.25 \times 10^4 + 49.28n + 0.12 \times 10^{-5}n^2 - 1.43 \times 10^{-9}n^3$
Red	$y = 30.37 \times 10^4 + 32.92n + 0.26 \times 10^{-5}n^2 - 0.00019 \times 10^{-9}n^3$
Toon	$y = 27.28 \times 10^4 + 37.30n + 0.23 \times 10^{-5}n^2 - 1.93 \times 10^{-9}n^3$
Flat	$y = 32.82 \times 10^4 + 33.84n + 0.28 \times 10^{-5}n^2 - 2.15 \times 10^{-9}n^3$
Random Color	$y = 26.25 \times 10^4 + 38.42n + 0,20 \times 10^{-5}n^2 - 1.76 \times 10^{-9}n^3$
Simple Texture	$y = 24.51 \times 10^4 + 38.88n + 0,18 \times 10^{-5}n^2 - 1.65 \times 10^{-9}n^3$
CubeMap	$y = 29.87 \times 10^4 + 44.70n + 0.11 \times 10^{-5}n^2 - 1.28 \times 10^{-9}n^3$
Reflection	$y = 33.63 \times 10^4 + 57.31n - 9.18 \times 10^{-5}n^2 - 0.35 \times 10^{-9}n^3$

One devices. The Figure 19 compares the shaders related to the rendering process for the Nexus 4 and the iOS devices.

**Figure 17: Nexus 4 and HTC One comparison: vertex shader****Figure 18: Nexus 4 and HTC One comparison: fragment shader**

By the measurements and obtained equations, the device that had better performance, related to the rendering process, was the iPad Air, which is the device with better hardware configuration. And as it was shown in Section 4.1, the iPad Air was the device with better position in the benchmarking app. The device with worst performance was the Nexus 4 and this is consistent because it has the worst hardware configuration.

For the vertex shader, the Nexus 4 had better performance than the HTC One. On the other hand, for the fragment shader, the HTC one had better performance than the Nexus 4.

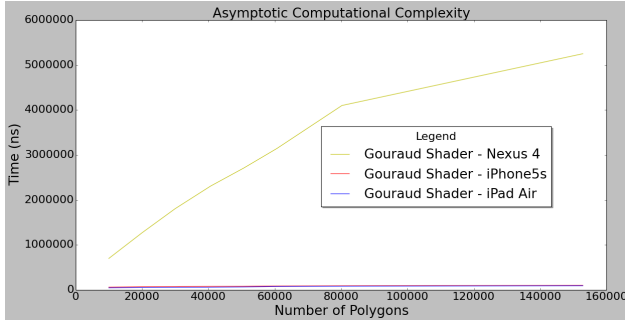


Figure 19: Nexus 4, iPhone 5s and iPad comparison: rendering process

5.3.2 Final Thoughts About the Equations

Through the results, it was revealed that both rendering process and fragment shader tended to present as asymptotic complexity a third degree function for any shader. However, even that the squared errors were smallest to a third degree function, the coefficients related to the n^2 term are very small, being of the order 10^{-7} , 10^{-8} and 10^{-9} . In case of 10^{-7} , by example, it will be added or subtracted one unit for each 100 million units of n (for a $y(n)$ function), which can be considered irrelevant.

This way, the curve related to the second degree function, even with bigger squared error, represents the reality of the shader better than the third degree function. Then, the asymptotic complexity of the fragment shader and of the rendering process can be considered $O(n^2)$. The analysis done to the third degree equations is still valid to the second degree equations.

5.4 Estimates in Production Environments

In the gaming industry, the metric commonly used to determine the performance of a game is the FPS (Frames Per Second). This metric represents the number of images rendered per second. This way, it's also possible to convert the obtained results in this work to this metric, like is shown in the Equation 11, where t is the time in seconds (the metric used for the rendering process).

$$FPS = \frac{1}{t} \quad (11)$$

It's also possible to obtain this time metric based on the number of instructions per second (the metric used for the vertex and fragment shaders). The Equation 12 shows how to do this conversion, in which is necessary to use the tool Adreno Profiler to get the number of instructions for one frame.

$$t = \frac{I_F}{I_{PS}} \quad (12)$$

where I_F is the number of instructions for one frame and I_{PS} is the number of instructions per second.

Before converting the metric to frames per second, for the rendering process, it was added to the t variables the time spent by the other functions in the OpenGL ES. These are the functions used for one frame in the draw call, that doesn't vary with the number of polygons, like the function that sets the background color, by example.

In Android devices, these times were obtained with the same OpenGL ES extension used before. In iOS devices, they were obtained with the Instruments tool, that informs the time spent by each function.

The Table 6 presents the converted results in frames per second, taking the Gouraud Shader as example, for the Nexus 4 device. An important observation is that this measures doesn't include the other factors present in a real production environment, like input events and physics, by example.

Number of Polygons	Time Spent (s): glDrawArrays	Time Spent (s): Other Functions	FPS
10,000	0.000698	0.0000140	1,405
20,100	0.00127	0.0000140	779
30,000	0.00181	0.0000140	548
40,678	0.00231	0.0000140	430
50,679	0.00271	0.0000140	367
60,662	0.00315	0.0000140	316
80,256	0.00410	0.0000140	243
152,840	0.00525	0.0000140	190

Table 6: Estimated FPS

5.5 Experimental Process of Estimation of Asymptotic Complexity

The process used in this work for empirically estimate the computational asymptotic complexity is represented in Figure 20. The step Implement Shaders can be done taking as base the implemented code in this work. It's just necessary to inherit from the Shader class and implementing its abstract methods. The step Measure Performance is done manually, depending on the GPU profiler needed. The steps Plot Charts, Adjust Curves and Obtain Equations can be done by the implemented tool.

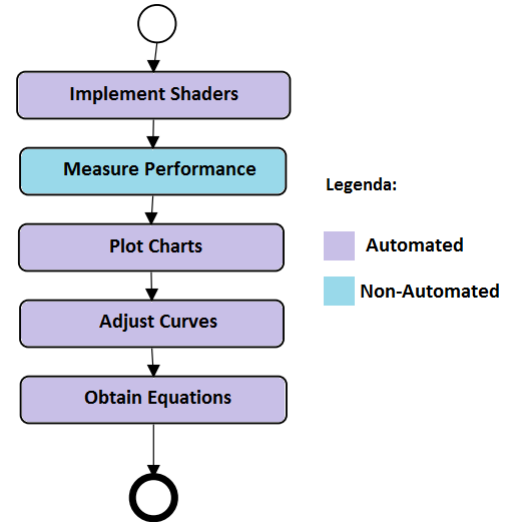


Figure 20: Asymptotic complexity process

6 Discussion and Future Work

Through the experiments, it was revealed that the asymptotic complexity behaved linearly for the vertex shader. This happened independently of the shader used. This way, all implemented vertex shaders have the same asymptotic complexity. But the equations for each one have different coefficients, that can determine which shader has better or worse performance.

Analyzing the theory about the OpenGL rendering process for the vertex shader, it can be seen that this result is consistent. The vertex shader program is used for each vertex, then the asymptotic complexity is linear, taking the number of vertices as input. So, the flow of the shader's execution can be represented by the Code 2.

Code 2: Representation of the vertex shader execution

```

for (int i = 0; i < vertexBuffer.length; i++)
{
    executeVertexShader(vertexBuffer[i]);
}
  
```

The rendering process and the fragment shader tended to have as asymptotic complexity a polynomial of second degree. The Code 3 shows a generic flow representation of the fragment shader execution.

Code 3: Representation of the fragment shader execution

```

triangleStream = Mesh.triangles;

for(int i = 0; i < triangleStream.length; i++)
{
    fragmentStream = triangleStream[i].fragments;

    for(int j = 0; j < fragmentStream.length; j++)
    {
        executeFragmentShader(fragmentStream[i]);
    }
}

```

As is explained in OpenGL's documentation² for each primitive of the mesh, it's generated the fragments (candidates for pixels). For each fragment, the horizontal and vertical orientations of the screen are traversed (being a matrix).

This way, the function `executeFragmentShader(fragment)` assigns to the fragment a color and a depth value (this values will be used in the last steps of the rendering process to discard some fragments). The quadratic asymptotic complexity probably is associated with the color attribution (which traverses a matrix, that is quadratic).

Besides, the obtained results are not obvious, because when a shader program is analyzed it induces the programmer to think that its asymptotic complexity is constant, which this work showed that it isn't. An example of a simple vertex shader is shown in Code 4.

Code 4: Example of vertex shader

```

uniform mat4 uMVPMatrix;
attribute vec4 aPosition;

void main() {
    gl_Position = uMVPMatrix * aPosition;
}

```

Since all the shaders (of the same type) present the same asymptotic complexity, a way to compare their performance is by their equations and coefficients. This analysis can be done related to the entire rendering process and only specifically to the vertex and fragment shaders. The comparison can be done to different shaders and to the same shader, to see if it was optimized or not. Another possible comparison is between devices, as was done in this work, which the iPhone 5s, that is from a more recent smartphone generation, had better performance than the Nexus 4.

Another important contribution was the automation of most of the asymptotic complexity analysis, like the shader implementation basis and curve adjustments. Like this, such a procedure can be reproduced quickly and reliably. As future work, would be interesting to implement more shaders, specially for iOS platform and also compare more devices.

References

- ARNAU, J., PARCERISA, J., AND XEKALAKIS, P. 2013. Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems. *International Conference on Supercomputing* (June), 37–46.
- COMPAQ COMPUTER CORPORATION. 1999. *Graphics Performance: Measures, Metrics and Meaning*.
- DROZDEK, A. 2002. *Estrutura de Dados e Algoritmos em C++*, 2 ed. Cengage Learning, So Paulo, So Paulo.
- GUHA, S. 2011. *Computer Graphics Through OpenGL*, 1 ed. CRC Press, Boca Raton, Florida.
- LEITHOLD, L. 1994. *O Clculo com Geometria Analitica*, 3 ed. Harbra, So Paulo, So Paulo.
- MOLLER, T. A., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering*, 2 ed. CRC Press, Boca Raton, Florida.
- N., B., AND S., A. 2007. Criando efeitos fotorealistas e no-fotorealistas para jogos. *SBGames*.
- N., G., B., R., V., A., R., A., AND F., B. 2011. Ganho de performance e economia de largura de banda com o uso do tessellator. *SBGames*.
- RORRES, A. 2001. *lgebra Linear com Aplicaes*, 8 ed. Bookman, Porto Alegre, Rio Grande do Sul.
- SANDBERG, R., AND ROLLINS, M. 2013. *The Business of Android Apps Development*, 2 ed. Apress, New York, New York.
- SHERROD, A. 2011. *Game Graphics Programming*, 1 ed. Course Technology, Boston, Massachusetts.
- SINTHANAYOTHIN, C., WONGWEAN, N., AND BHOLSITHI, W. 2012. Interactive virtual 3d gallery using motion detection of mobile device. *International Journal of Advancements in Computing Technology* 4, 7, 239–250.

²http://www.opengl.org/wiki/Fragment_Shader