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

Alex S. C. Lima, Edson A. C. Junior
Gama College
University of Brasilia
Brasilia, Brazil
campelo.al1@gmail.com, prof.edson.alves.costa@gmail.com

***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; asymptotic complexity.**

I. Introduction

Graphics in games are so important that can determine the game's failure or success [1]. Thus, the creation of three-dimensional scenes, using mobile devices, is becoming more and more usual and realistic [2]. However, there are 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 [3].

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 sold[1] and the daily activation devices using Android platform is approximately 1.5 million [4].

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).

II. 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 [5], where were presented diverse realistic and non-realistic techniques used in games in the last years. To achieve those effects, the programmable rendering pipeline was used, specifically the vertex and fragment shaders.

In [6] is shown 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. 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. However, there is not an unique standard to graphics benchmark in mobile devices, and several approaches has been proposed to the present day (i.e., [7], [8], [9], [10]).

The shader performance analysis were already done in [11], 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, what, in practical terms, is equal to increase the number of polygons. The chosen metric to analyze the performance was the frames per second.

[1] http://www.imore.com/more-800-million-ios-devices-sold

III. 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.

A. Shader and OpenGL ES

Shading is the process of using an equation for computing the surface behavior of an object [12]. Shader algorithms 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 [13], 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.

B. Fundamentals of Physics and Mathematics for Shaders Implementation

The visual effects - created through shaders - are representations of physical events and may assign different materials to objects and light effects, for instance. Thus, this subsection presents some concepts necessary for understanding the implemented shaders.

1) Gouraud and Phong Shading: The computer calculations of the light at the vertices, followed by linear interpolation of the results is known as Gouraud shading, created by Henri Gouraud. Its vertex shader calculates the intensity of light at each vertex and the results are interpolated. Then, the fragment shader propagates this value for the next steps in the rendering process.

In Phong shading, firstly, the normal values of the primitives are interpolated. Then, the light values are calculated for each fragment, using the interpolated normal values. As said by [13], the light intensity on a surface point is calculated as shown in Equation 1

$$I_T = I_A + \sum I_D + I_S, \tag{1}$$

where I_T is the total illumination, I_A is the ambient illumination, I_D is the diffuse illumination and I_S is the specular illumination.

The ambient reflection intensity comes from all directions and when it reaches the surface, it also spreads out in all directions, having the same intensity for all points. It can be calculated according to Equation 2

$$I_A = K_A L_A, \tag{2}$$

where I_A is the intensity of ambient light, K_A is the surface's coefficient of ambient reflection and L_A is the component of the intensity of ambient light.

The intensity of the diffuse reflection comes from one direction and when it reaches a surface, spreads equally in all directions. It can be calculated according to Equation 3

$$I_D = K_D L_D (\vec{l} \cdot \vec{n}), \tag{3}$$

where I_D is the intensity of the diffuse reflection, K_D is the diffuse reflection coefficient of the surface, L_D is the component of intensity of the diffuse light, $\vec{l}$ is the light source and $\vec{n}$ is the point of interest.

The specular light comes from one direction and reflects like a mirror, which the incident angle is equal to the reflection angle. It can be calculated as shown in Equation 4

$$I_S = K_S L_S (\vec{r} \cdot \vec{v})^s, \tag{4}$$

where I_S is the intensity of the specular reflection, K_S is the specular reflectance of the surface, L_S is the intensity of the specular component of light, $\vec{r}$ is the direction of the reflection, $\vec{v}$ is the observer vector and s is the specular exponent.

2) Toon Shading: The Toon shading aims to simulate the effect of a cartoon, whose main characteristic is the uniform appearance of colors on the surface of the objects. This can be done by mapping the light intensity ranges for specific colors, in order to give a less realistic lighting and nearest to the one used in cartoons. The light intensity can be calculated according to Equation 5

$$I_L = \frac{\vec{l_d} \cdot \vec{n}}{\|\vec{l_d}\| \|\vec{n}\|}, \tag{5}$$

where I_L is the light intensity, $\vec{l_d}$ is direction of the light's vector and $\vec{n}$ is the normal vector.

Because $\vec{l_d}$ and $\vec{n}$ are normalized vectors, the calculation can be simplified according Equation 6.

$$I_L = \vec{l_d} \cdot \vec{n} \tag{6}$$

3) Reflection Shading: The reflection effect is obtained by using the technique called cube mapping (a type of environment mapping), which uses the six faces of a cube as the map shape. So this shader reflects this texture, and one example of a texture to be used for cube mapping is shown in Fig. 1.

So the idea is to get the reflected vector of the camera position to the object, based on the surface normal. This

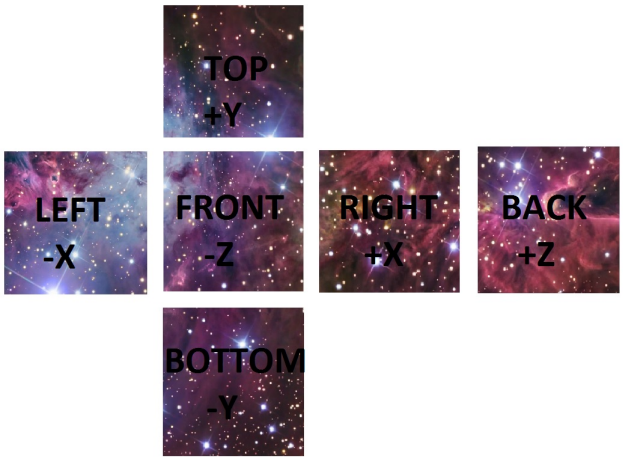

Figure 1. *Cube Map*

vector is used to determine the color of the fragment, based on the texture.

C. 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) [14]. 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. For instance, the Equation 7

$$y = n^2 + 10n + 1000 \tag{7}$$

could be approximated by Equation 8.

$$y \approx n^2 \tag{8}$$

D. 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 $y = a + bx$,in most cases the points in the set aren't collinear. In this situation, as said in [15], 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 9

$$v = (M^T M)^{-1} M^T y, \tag{9}$$

where

$$M = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, \quad v = \begin{bmatrix} a \\ b \end{bmatrix} \text{ e } \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{10}$$

The adjustment to a second and third degree functions is similar, but the M matrix is redefined to Equation 11

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix} \tag{11}$$

and to Equation 12, respectively.

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_1^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 \end{bmatrix} \tag{12}$$

The exponential adjustment is a little bit different. [16] says that the exponential function can be represented as Equation 13,

$$y = ce^{-kt} \tag{13}$$

where e, c, k are constants. Applying the logarithmic function on both sides, the Equation 14 is obtained

$$\ln y = \ln c + \ln e^{-kt} \tag{14}$$

and it can be simplified as Equation 15 (where $\bar{b}$ is a new constant).

$$\bar{y} = \bar{a} + \bar{b}t \tag{15}$$

This equation is equivalent to a linear equation and the linear least squares method can be applied. The final values of the $\bar{a}$ and $\bar{b}$ coefficients determine the c and k parameters through the relationships shown in Equation 16

$$c = e^{\bar{a}} \text{ e } \bar{b} = -k \tag{16}$$

IV. Materials and Methods

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

A. 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 I shows the used devices, which are equipment with different resolutions and hardware configurations. The benchmark app called 3D Mark was used to compared

Table I
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 II
BENCHMARK

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

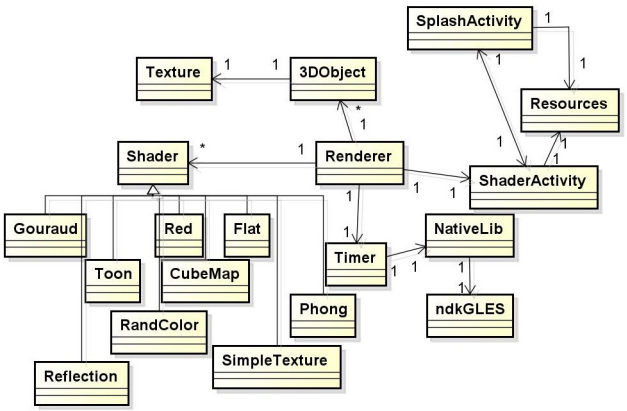

Figure 2. Android implementation: class diagram

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 II.

B. 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 Fig. 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.

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 are two Activity classes: Shader Activity and Splash Activity (Fig. 3).

The Splash Activity is responsible for the visualization of the loading screen while the necessary resources – like

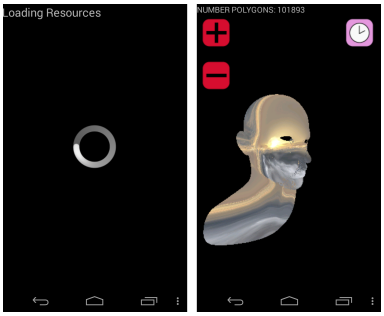

Figure 3. Shader Activity and Splash Activity

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.

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.

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

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 variable 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 Fig. 4.

5) Phong Shader: The vertex and fragment shaders of Phong shading implement the theory described on Section III-B1, where the normal values of the primitives are interpolated on the vertex shader, and then the lightning

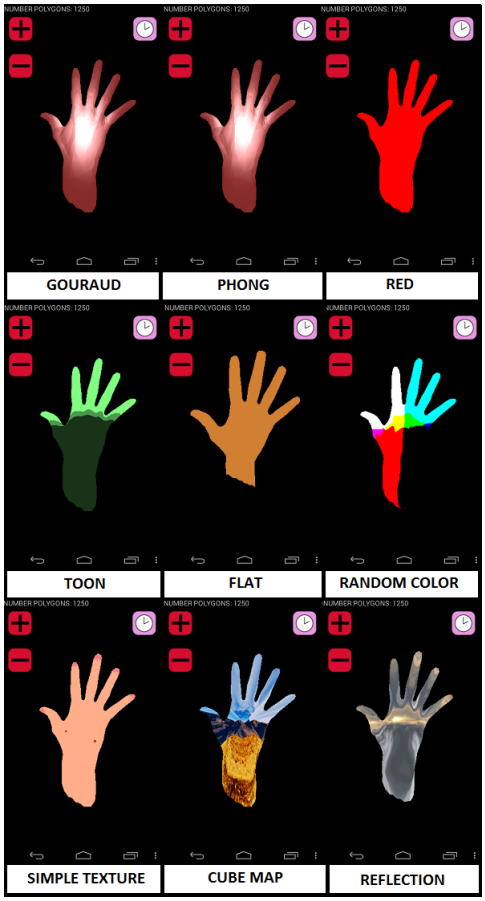

Figure 4. Implemented Shaders

calculations are done in the fragment shader. The Listing 1 and the Listing 2 show the definitions of the vertex and fragment shaders, respectively.

Listing 1. Phong Shader: vertex shader

```
uniform mat4 uMVPMatrix;
uniform mat4 normalMatrix;
uniform vec3 eyePos;
attribute vec4 aPosition;
attribute vec3 aNormal;
uniform vec4 lightPos;
uniform vec4 lightColor;
uniform vec4 matAmbient;
uniform vec4 matDiffuse;
uniform vec4 matSpecular;
uniform float matShininess;
varying vec3 vNormal;
varying vec3 EyespaceNormal;
varying vec3 lightDir, eyeVec;

void main()
{

  EyespaceNormal = vec3(normalMatrix
        * vec4(aNormal, 1.0));
  vec4 position = uMVPMatrix * aPosition;
  lightDir = lightPos.xyz - position.xyz;
  eyeVec = -position.xyz;

  gl_Position = uMVPMatrix * aPosition;
}
```

Listing 2. Phong Shader: fragment shader

```
precision mediump float;
varying vec3 vNormal;
varying vec3 EyespaceNormal;
uniform vec4 lightPos;
uniform vec4 lightColor;
uniform vec4 matAmbient;
uniform vec4 matDiffuse;
uniform vec4 matSpecular;
uniform float matShininess;
uniform vec3 eyePos;
varying vec3 lightDir, eyeVec;

void main()
{
  vec3 N = normalize(EyespaceNormal);
  vec3 E = normalize(eyeVec);
  vec3 L = normalize(lightDir);
  vec3 reflectV = reflect(-L, N);

  //Ambient Lighting
  vec4 ambientTerm;
  ambientTerm = matAmbient * lightColor;

  //Diffuse Lighting
  vec4 diffuseTerm = matDiffuse *
        max(dot(N, L), 0.0);

  //Specular Lighting
  vec4 specularTerm = matSpecular *
        pow(max(dot(reflectV, E), 0.0),
         matShininess);

  gl_FragColor =  ambientTerm +
         diffuseTerm + specularTerm;

}
```

6) Gouraud Shader: The Gouraud Shader, as the Phong Shader, also implements the theory described on Section III-B1, except for the lightning calculations. They are done on the vertex shader and the final values are interpolated and used by the fragment shader, as shown on Listing 3 and Listing 4.

Listing 3. Gouraud Shader: vertex shader

```
uniform mat4 uMVPMatrix;
uniform mat4 normalMatrix;
// eye pos
uniform vec3 eyePos;
// position and normal of the vertices
attribute vec4 aPosition;
attribute vec3 aNormal;
// lighting
uniform vec4 lightPos;
uniform vec4 lightColor;
// material
```

```
uniform vec4 matAmbient;
uniform vec4 matDiffuse;
uniform vec4 matSpecular;
uniform float matShininess;
// color to pass on
varying vec4 color;

void main() {

  vec3 eP = eyePos;
  vec4 nm = normalMatrix *
  vec4(aNormal, 1.0);
  vec3 EyespaceNormal = vec3(uMVPMatrix *
        vec4(aNormal, 1.0));
  // the vertex position
  vec4 posit = uMVPMatrix * aPosition;
  // light direction
  vec3 lightDir = lightPos.xyz - posit.xyz;
  vec3 eyeVec = -posit.xyz;
  vec3 N = normalize(EyespaceNormal);
  vec3 E = normalize(eyeVec);
  vec3 L = normalize(lightDir);
  // Reflect the vector
  vec3 reflectV = reflect(-L, N);
  // Ambient Lightning
  vec4 ambientTerm;
  ambientTerm = matAmbient * lightColor;
  // Diffuse Lightning
  vec4 diffuseTerm = matDiffuse *
        max(dot(N, L), 0.0);
  // Specular Lightning
  vec4 specularTerm = matSpecular *
        pow(max(dot(reflectV, E), 0.0),
        matShininess);
  color =  ambientTerm + diffuseTerm
        + specularTerm;

  gl_Position = uMVPMatrix * aPosition;
}
```

Listing 4. Gouraud Shader: fragment shader

```
precision mediump float;
// the color
varying vec4 color;
void main() {
        gl_FragColor = color;
}
```

7) Red Shader: This is a simple shader that sets the vertex position and the fragment color to red, as shown on Listing 5 and Listing 6.

Listing 5. Red Shader: vertex shader

```
uniform mat4 uMVPMatrix;
attribute vec4 aPosition;

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

Listing 6. Red Shader: fragment shader

```
void main() {
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

8) Toon Shader: The Toon shader calculates the lightning intensity per vertex (as presented on Section III-B2), to choose a pre-defined color. The Listing 7 and Listing 8 show this calculation.

Listing 7. Toon Shader: vertex shader

```
uniform vec3 lightDir;
uniform mat4 uMVPMatrix;
attribute vec3 aNormal;
attribute vec4 aPosition;
varying float intensity;

void main()
{
  intensity = dot(lightDir,aNormal);
  gl_Position = uMVPMatrix * aPosition;
}
```

Listing 8. Toon Shader: fragment shader

```
varying float intensity;

void main()
{
  vec4 color;

  if (intensity > 0.95)
    color = vec4(0.5,1.0,0.5,1.0);
  else if (intensity > 0.5)
    color = vec4(0.3,0.6,0.3,1.0);
  else
    color = vec4(0.1,0.2,0.1,1.0);

  gl_FragColor = color;
}
```

9) Flat Shader: The idea of the Flat Shader is to transform a three-dimensional object into a two-dimensional one. To do this, the z coordinate is defined as zero, as shown on Listing 9. Its fragment shader (Listing 10) just defines a color to the the fragment.

Listing 9. Flat Shader: vertex shader

```
uniform mat4 uMVPMatrix;
attribute vec4 aPosition;

void main()
{

  vec4 v = aPosition;
  v.z = 0.0;
  gl_Position = uMVPMatrix * v;

}
```

Listing 10. Flat Shader: fragment shader

```
void main()
{
  gl_FragColor = vec4(0.82, 0.50, 0.20, 1.0);
}
```

10) Random Color Shader: The Random Color shader determines the color of the fragment randomly, based on a mathematical calculation. This calculation is done on the vertex shader (Listing 11) and the final value is given to the fragment shader. Each color component is calculated using as parameter a coordinate (x, y ou z) given to the function `random(vec2 v)`, which returns a random value based on the given coordinate. In the Listing 12 the color of the fragment is set.

Listing 11. Random Color Shader: vertex shader

```
uniform mat4 uMVPMatrix;
attribute vec4 aPosition;
varying vec4 color;

float random( vec2 v){
  // e^pi (Gelfond's constant)
  const vec2 r = vec2(23.1406926327792690,
  // 2^sqrt(2) (GelfondSchneider constant)
                      2.6651441426902251);
  return mod( 123456789., 1e-7 +
        256. * dot(v,r) );
}

void main()
{
   vec2 r = vec2(aPosition.x, aPosition.z);
   vec2 g = vec2(aPosition.y, aPosition.x);
   vec2 b = vec2(aPosition.z, aPosition.y);
   color = vec4(random(r),random(g),
        random(b),1.0);
   gl_Position = uMVPMatrix * aPosition;

}
```

Listing 12. Random Color Shader: fragment shader

```
varying mediump vec4 color;

void main()
{
  gl_FragColor = color;
}
```

11) Simple Texture Shader: The vertex shader of the Simple Texture Shader stores the texture coordinates into a variable (Listing 13), and gives it to the fragment shader. On Listing 14, the fragment shader uses these coordinates and applies it into a texture by using the function `texture2D()`.

Listing 13. Simple Texture Shader: vertex shader

```
uniform mat4 uMVPMatrix;
attribute vec4 aPosition;
attribute vec2 textCoord;
varying vec2 tCoord;

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

Listing 14. Simple Texture Shader: fragment shader

```
varying vec2 tCoord;
uniform sampler2D texture;

void main()
{
  gl_FragColor = texture2D(texture,tCoord);
}
```

12) CubeMap Shader: The vertex shader of the CubeMap Shader just defines the vertex position (Listing 15). The fragment shader (Listing 16), uses the `textureCube(samplercube s, vec3 coord)` function, which receives as parameters the normal vector and the texture to be mapped.

Listing 15. CubeMap Shader: vertex shader

```
attribute vec4 aPosition;
attribute vec3 aNormal;
varying vec3 v_normal;
uniform mat4 uMVPMatrix;

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

Listing 16. CubeMap Shader: fragment shader

```
precision mediump float;
varying vec3 v_normal;
uniform samplerCube s_texture;
void main()
{
  gl_FragColor = textureCube( s_texture,
        v_normal );
}
```

13) Reflection Shader: The Reflection Shader implements the theory described on Section III-B3. Its vertex shader is responsible for defining the vertex position, as shown on Listing 17. Besides, it also declares two vectors of the type varying (which gives the variables' values to the fragment shader), that are related to the vector of the camera's direction and normal. In the fragment shader these vectors are used to find the reflection vector by the `reflect` function. The reflection vector is used on the `textureCube(samplercube s, vec3 coord)`, which determines the fragment's color based on this vector and on an image.

Listing 17. Reflection Shader: vertex shader

```
attribute vec4 aPosition;
attribute vec3 aNormal;

varying vec3 EyeDir;
varying vec3 Normal;

uniform mat4 MVMatrix;
uniform mat4 uMVPMatrix;
uniform mat4 NMatrix;
```

```
10
11 void main()
12 {
13   gl_Position = uMVPMatrix * aPosition;
14   EyeDir=vec3(MVMatrix*aPosition);
15   Normal = mat3(NMatrix) * aNormal;
16 }
```

.

Listing 18. Reflection Shader: fragment shader

```
1  varying vec3 EyeDir;
2  varying vec3 Normal;
3  uniform samplerCube s_texture;
4
5  void main()
6  {
7    mediump vec3 reflectedDirection =
8          normalize(reflect(EyeDir,
9          normalize(Normal)));
10
11   reflectedDirection.y = -reflectedDirection.y;
12   gl_FragColor = textureCube( s_texture,
13         reflectedDirection);
14 }
```

14) 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.

C. iOS Implementation

The structure of the code on iOS platform is similar to the Android, as is shown in Fig. 5. It follows the Model-View-Controller pattern, which the controller is responsible for the integration between the Shader, 3DObject classes and the view RendererView.

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 Fig. 6.

D. 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.

1) Rendering Process: In Android platform, as mentioned in Section IV-B14, an OpenGL ES extension was used to get the rendering process time, done by the `glDrawArrays()` function. In iOS platform, the module

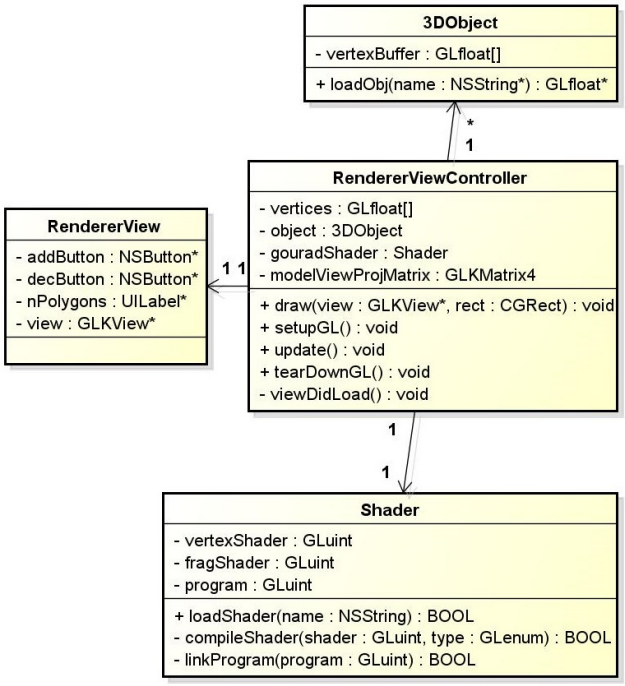

Figure 5. iOS implementation: class diagram

Figure 6. Gouraud Shader on iOS platform

– 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 device, because the extension wasn't available for this Android device.

2) Vertex and Fragment Shaders: The vertex and fragment shaders measurements were only possible to do in Android devices. The reason is because the Instruments module of Xcode – in iOS implementation – doesn't exhibit any information about them. Then, the tool used to collect the measures for Android devices was the Adreno Profile, because the GPUs of these devices are Adreno.

The chosen metrics were instructions per second per

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

3) 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) Automation of Curves Adjustments: To do the curve adjustment, it 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 Listing 19 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.

Listing 19. Command-lines

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

The Fig. 7 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 data and the LeastSquares calculates these adjustments and their errors.

In Fig. 8 the result of the tool for the rendering process is presented, which is composed by four screens. The first one (top-down) 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.

In Fig. 9 and Fig. 10 the results of the tool for the vertex and fragment shaders are presented, which are four screens with the adjustments as well. At the end, the program shows the equations related to each adjustment and their errors.

V. 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).

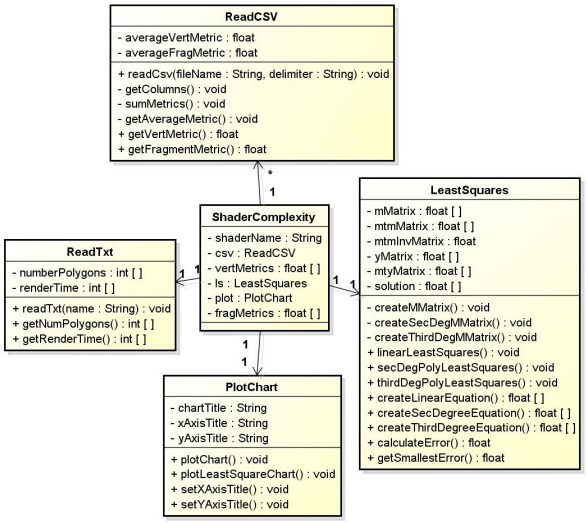

Figure 7. Tool implementation: class diagram

A. 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 Fig. 11 shows the charts related to the vertex shader of all implemented shaders.

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 Fig. 12 and Fig. 13.

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 (Fig. 14).

B. 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 Fig. 15 shows these curves for the iPhone 5s and for the iPad Air.

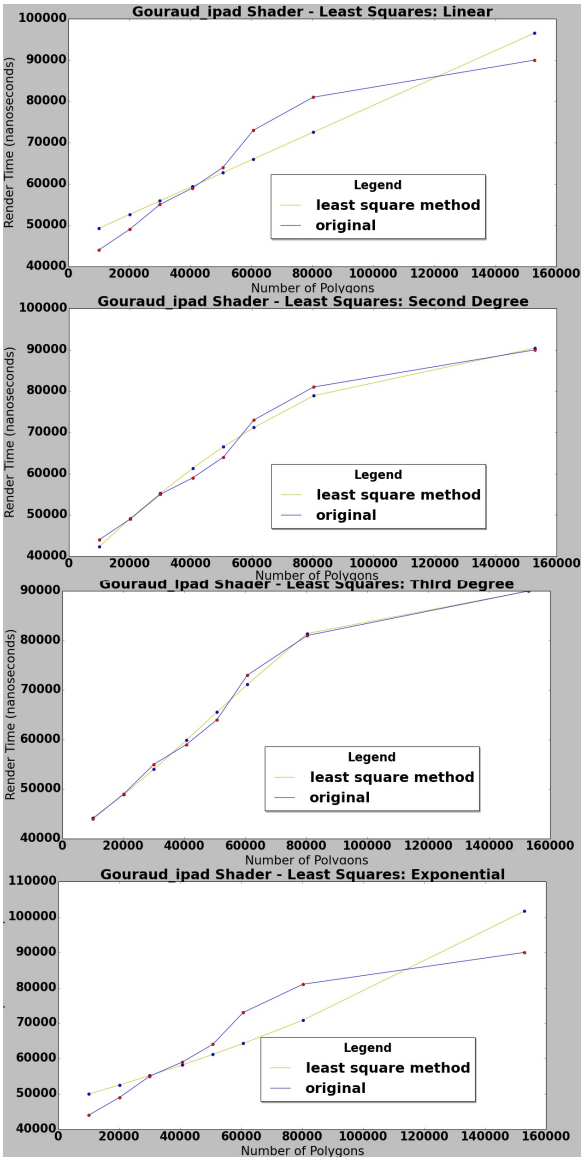

Figure 8. Automatic Adjustments: rendering process

C. 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 III, Table IV and Table V. 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 they had greater x^2 and x^3 coefficients.

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

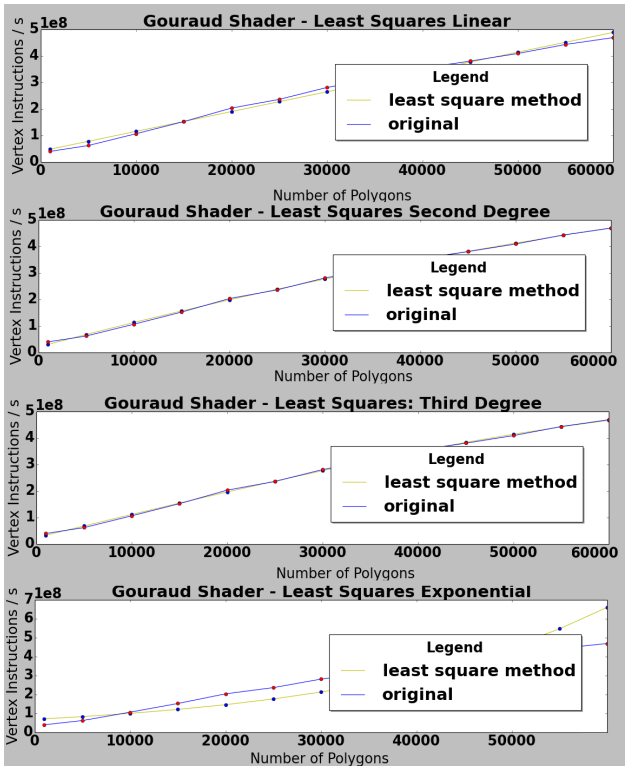

Figure 9. Automatic Adjustments: vertex shader

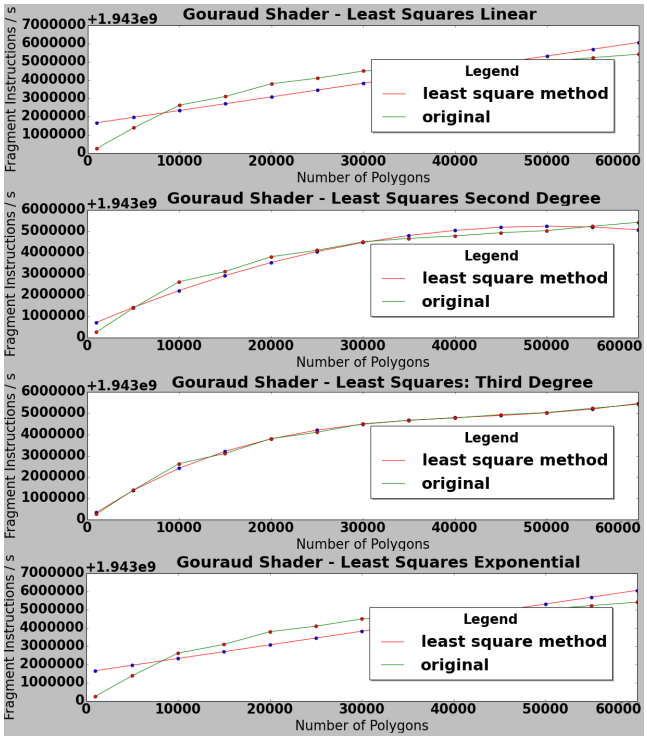

Figure 10. Automatic Adjustments: fragment shader

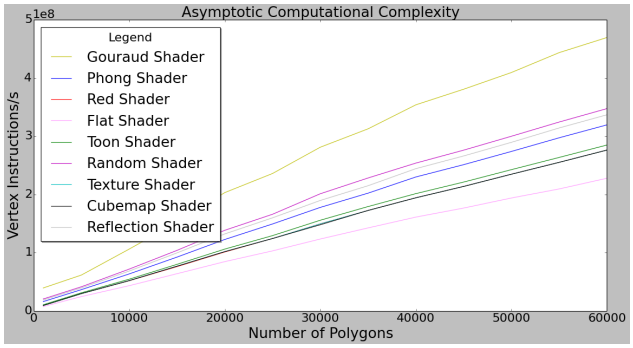

Figure 11. Vertex shader: curves comparison

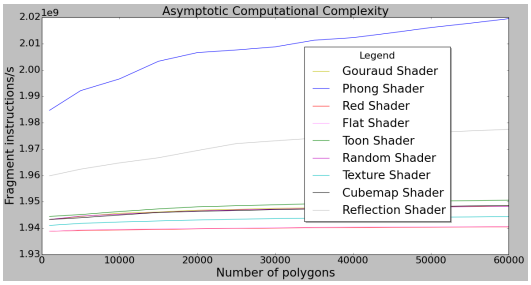

Figure 12. Fragment shader: curves comparison

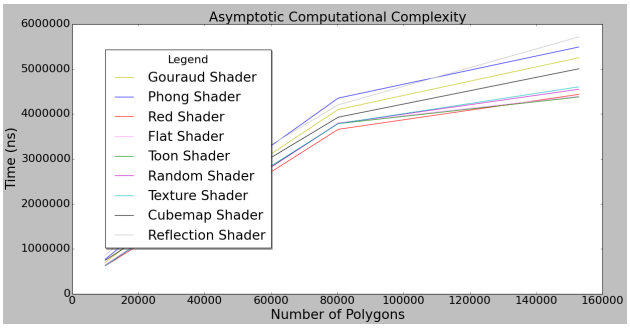

Figure 13. Rendering process: curves comparison

Table III
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	$y = 18.03 \times 10^6 + 5470.95n$

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

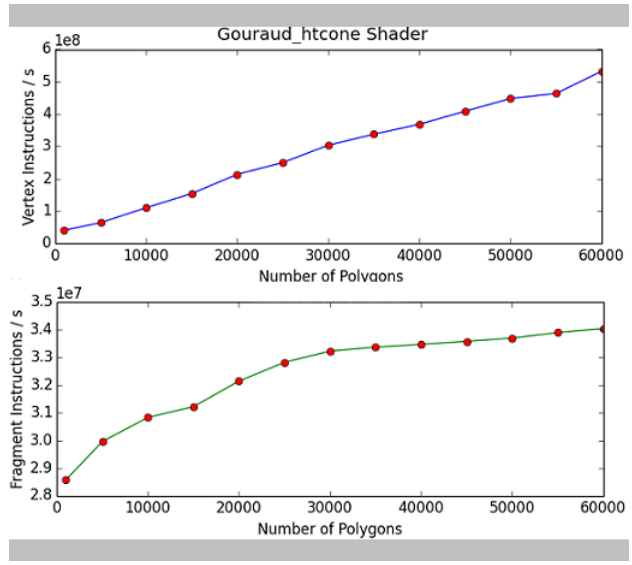

Figure 14. HTC One device

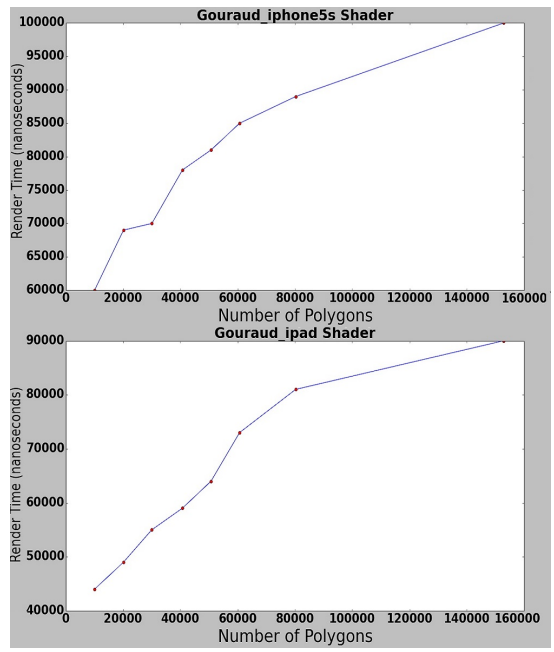

Figure 15. iOS devices

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

Table IV
EQUATIONS RELATED TO THE FRAGMENT SHADER

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

Table V
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	$y = 33.63 \times 10^4 + 57.31n - 9.18$ $\times 10^{-5}n^2 - 0.35 \times 10^{-9}n^3$

number of instructions per second for 60,000 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 asymptotic complexity $O(n^3)$ and the vertex shader, $O(n)$, influencing this worst outcome.

1) Devices Comparison: With the obtained curves and equations, it was also possible to compare the devices. The

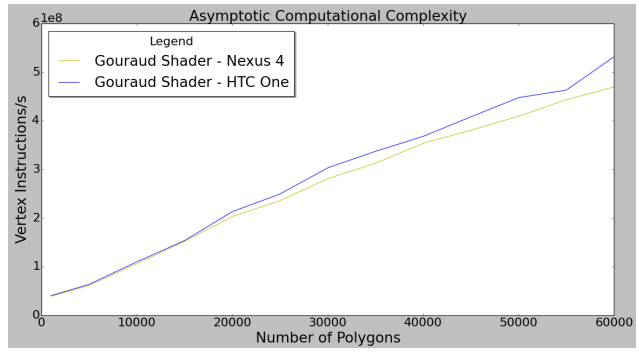

Figure 16. Nexus 4 and HTC One comparison: vertex shader

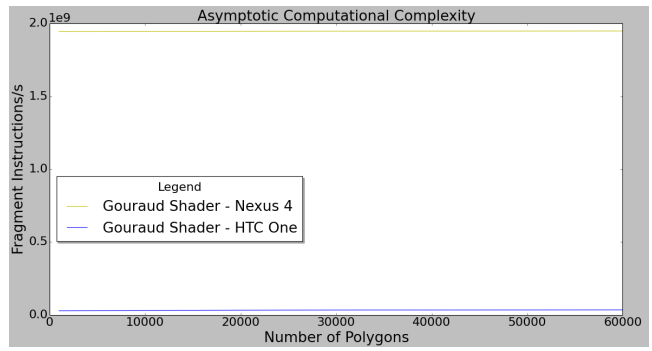

Figure 17. Nexus 4 and HTC One comparison: fragment shader

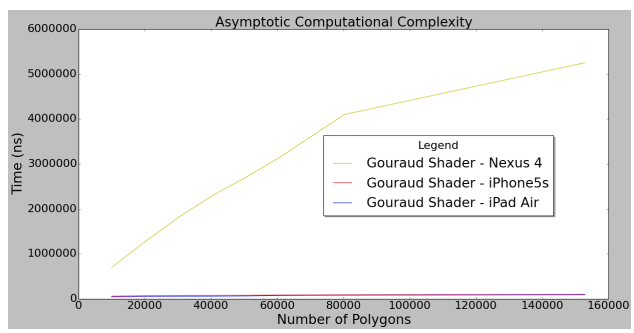

Figure 18. Nexus 4, iPhone 5s and iPad comparison: rendering process

Fig. 16 and Fig. 17 shows the curves related to the vertex and fragment shaders for the Nexus 4 and HTC One devices. The Fig. 18 compares the shaders related to the rendering process for the Nexus 4 and the iOS devices.

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 IV-A, the iPad Air was the device with better position in the benchmark 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.

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^3 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.

D. 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 Equation 17, where t is the time in seconds (the metric used for the rendering process).

$$FPS = \frac{1}{t} \tag{17}$$

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 18 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.

$$t = \frac{I_F}{I_S} \tag{18}$$

where I_F is the number of instructions for one frame and I_S is the number of instructions per second.

Before converting the time metric to frames per second, for the rendering process, it was added to the t variable, 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 VI presents the converted results in frames per second, taking the Gouraud Shader as example, for the Nexus 4 device. An important observation is that these measures don't include the other factors present in a real production environment, like input events and physics, by example.

Table VI
ESTIMATED FPS

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

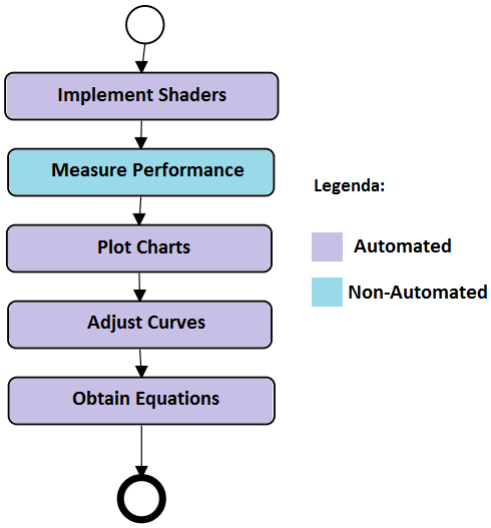

Figure 19. Experimental Process

E. Experimental Process of Estimating the Asymptotic Complexity

The process used in this work to estimate, empirically, the computational asymptotic complexity of shaders is represented in Fig. 19.

The step Implement Shaders can be done, for iOS and Android devices, taking as base the implemented code in this work. It's just necessary to inherit from the Shader class and to implement 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. These codes can be found on the remote repository[2].

VI. DISCUSSION AND FUTURE WORK

Through the experiments, it was revealed that the asymptotic complexity behaved linearly for the vertex shader.

[2] *https://github.com/campeloal*

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 its asymptotic complexity is linear, taking the number of vertices as input. So, the flow of the shader's execution can be represented by the Listing 20.

Listing 20. 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 Listing 21 shows a generic flow representation of the fragment shader execution.

Listing 21. Representation of the fragment shader execution

```
triangleStream = Mesh.triangles;

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

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

}
```

As explained in OpenGL's documentation [3] 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 `executeFragShader(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 source code 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 Listing 22.

Listing 22. 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 it was done in this work. The iPhone 5s, that is from a more recent smartphone generation, had better performance than the Nexus 4.

As seen on this work, when rendering objects in a scene with a shader, different performances were obtained for each device. This way, these performance differences could influence the user experience while playing a game. The rendering in some devices are expected to be smoother than in others. This is because the update frame rates are affected differently, depending on the hardware configuration used.

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

[1] A. Sherrod, *Game Graphics Programming*, 1st ed. Boston, Massachusetts: Course Technology, 2011.

[2] C. Sinthanayothin, N. Wongwean, and W. Bholsithi, "Interactive virtual 3d gallery using motion detection of mobile device," *International Journal of Advancements in Computing Technology*, vol. 4, no. 7, pp. 239–250, 2012.

[3] J. Arnau, J. Parcerisa, and P. Xekalakis, "Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems," *International Conference on Supercomputing*, pp. 37–46, Jun. 2013.

[4] R. Sandberg and M. Rollins, *The Business of Android Apps Development*, 2nd ed. New York, New York: Apress, 2013.

[5] B. Evangelista and A. Silva, "Criando efeitos fotorealistas e no- fotorealistas para jogos," *SBGames*, 2007.

[6] *Graphics Performance: Measures, Metrics and Meaning*, Compaq Computer Corporation, 1999.

[3] *http://www.opengl.org/wiki/Fragment_Shader*

[7] J.-H. Nah, Y. Suh, and Y. Lim, "L-bench: An android benchmark set for low-power mobile gpus," *Computers & Graphics*, vol. 61, pp. 40–49, 2016.

[8] X. Ma, Z. Deng, M. Dong, and L. Zhong, "Characterizing the performance and power consumption of 3d mobile games," *Computer*, vol. 46, no. 4, pp. 76–82, Apr. 2013. [Online]. Available: http://dx.doi.org/10.1109/MC.2012.190

[9] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 37–46. [Online]. Available: http://doi.acm.org/10.1145/2464996.2464999

[10] B. Johnsson and T. Akenine-Möller, "Measuring per-frame energy consumption of real-time graphics applications," *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 1, pp. 60–73, March 2014. [Online]. Available: http://jcgt.org/published/0003/01/03/

[11] G. Nunes, R. Braga, A. Valdetaro, A. Raposo, and B. Feijo, "Ganho de performance e economia de largura de banda com o uso do tessellator," *SBGames*, 2011.

[12] T. A. Moller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 2nd ed. Boca Raton, Florida: CRC Press, 2008.

[13] S. Guha, *Computer Graphics Through OpenGL*, 1st ed. Boca Raton, Florida: CRC Press, 2011.

[14] A. Drozdek, *Estrutura de Dados e Algoritmos em C++*, 2nd ed. Sao Paulo, Sao Paulo: Cengage Learning, 2002.

[15] A. Rorres, *Algebra Linear com Aplicacoes*, 8th ed. Porto Alegre, Rio Grande do Sul: Bookman, 2001.

[16] L. Leithold, *O Calculo com Geometria Analitica*, 3rd ed. Sao Paulo, Sao Paulo: Harbra, 1994.