An efficient architecture for automatic shaders management on virtual globes

Agustín Trujillo
CTIM, Universidad de Las Palmas de Gran Canaria
Islas Canarias, Spain
Email: atrujillo@dis.ulpgc.es

José P. Suárez
IUMA, Universidad de Las Palmas de Gran Canaria
Islas Canarias, Spain
Email: josepablo.suarez@ulpgc.es

José M. Santana
CTIM, Universidad de Las Palmas de Gran Canaria
Islas Canarias, Spain
Email: josemiguel.santana@ulpgc.es

Manuel de la Calle
I+D Department at IGO Software,
 Cáceres, Spain
Email: mdelacalle@igossoftware.es

Diego Gómez-Deck
I+D Department at IGO Software,
 Cáceres, Spain
Email: diego@consultar.com

Abstract—This paper presents the last advances in Glob3 Mobile, a multi-platform graphics engine for Virtual Globes oriented to any kind of user. More specifically, the document focuses on the possibilities that programmable graphic pipelines have to offer regarding the rendering of many kinds of geospatial symbology. This paper explains a few key aspects about what shading programs are, and their benefits compared to the fixed rendering pipeline paradigm. It is also discussed how to manage the programs running on GPU in an user-friendly way, and the architecture that allows to perform this management keeping a high performance during the rendering.

Keywords—OpenGL, virtual globe, programmable pipeline, GPU, shader, symbology, terrain

I. INTRODUCTION

At the time the Glob3 Mobile project started, mobile devices were a new platform full of opportunities. Developers had the chance to present their ideas in a new and very interactive way. Geospatial data viewers on these devices were not as widespread as they are today, regardless of some commercial products as Google Earth (released in 2005) that brought the attention of the general public to the capabilities of this software. There were few real alternatives when trying to develop mobile apps that were open, free, and versatile enough. The best example of a free multi-purpose system was the World Wind virtual globe by NASA 1, which does not run on platforms such as iOS due to its Java-oriented nature.

For these reasons, it was developed a new graphics engine oriented to geospatial data that allows to create mobile applications with a minimum effort. Nowadays, many apps are intended to run on multiple platforms. Therefore, the engine had to permit its users to develop in the language of their choice, providing a compatible version of the API. So Glob3 Mobile was focused on the three main mobile platforms that are, without doubt, Android, iOS and in a more general way, HTML5.

Besides the terrain rendering, virtual globes have become a good way to communicate data with a geographical component to the final user. Each kind of data could be rendered as a different symbol. There are endless ways that these symbols can be displayed, and modern GPUs allow to explicitly program how the final image is created (shading programs or shaders). The purpose of this document is to show how Glob3 Mobile has become a multi-shader software for users that are unaware of the existence of shaders.

II. THE GLOB3 MULTI-PLATFORM ARCHITECTURE

Multi-platform development can be a tough problem itself. The solution requires to achieve the same functionality on every device independently of the operating system. Besides, it would be desirable to perform only one implementation and delegate the conversion between platforms to an automated process.

Android 2 is an open source software project and operating system developed by Google for mobile devices. Android has increasingly gained popularity during the last years, overcoming its competitors in 2011. The next big competitor in the mobile OS market is iOS 3, intended for iPhone’s and iPad’s, created by Apple Inc. Finally, Glob3 Mobile could run in any web platform due to the new web development standards HTML 5 4, CSS 3 and WebGL 5. WebGL has been tested as a suitable platform for intensive 3D application such as videogames and, of course, large terrain rendering [1] [8]. This allows the user to create web apps able to run on any web browser compliant with the latest standards.

The followed scheme [9] starts writing the core code in C++ to obtain the iOS engine. It can be used an automatic converter to turn it into Java obtaining the Android core. Afterwards, the GWT tool (Google) will be able to convert it into the Javascript code. The parts of the code that are platform specific such as access to the sensors, the multitouch interaction, internet access and other resources that are platform-dependent should be encapsulated in common interfaces and implemented specifically in the native language of each operating system.

---

1 NASA World Wind http://worldwind.arc.nasa.gov/java/
2 Google’s Android http://www.android.com
3 Apple’s iOS http://www.apple.com/es/ios/
4 HTML5 by WC3 http://www.w3.org/TR/html5/
5 WebGL by Khronos http://www.khronos.org/webgl/
For the graphic part it has to be considered that WebGL is the Javascript counterpart of OpenGL for Embedded Systems (OpenGL ES) 2.0 [5]. Due to this fact, it has been created a facade to access in the same way OpenGL ES 2.0 in mobile OS and WebGL on the Web.

III. RENDERING SCENES WITH SYMBOLOGY

The Glob3 Mobile software permits the representation of terrain as a 3D model. This model is based on a geometrical definition of the planet, which can be flat, spherical or ellipsoidal, and is covered by imagery provided by online resources such as Web Map Services and others. Another tool for creating a much more realistic terrain is the use of a Digital Model of Terrain which allows the user to create a model that includes elevations.

From a merely rendering point of view terrain could be quite simple to draw. There are many ways in which terrain could be rendered [3]. The pieces of terrain are a grid of vertices with a single texture painted on as seen in Figure 1. The amount of triangles drawn on each frame is held constant using Level Of Detail techniques [2] which are discussed in detail in other papers. [10]

Soon the engine had to render much more than only terrain. For the sake of this paper it is called symbol to anything shown in the scene that represents data, from simple points or lines to complex 3D models. As a generic software, the Glob3 engine should provide to the user an understandable way of implementing the symbols that they need, to show their data in as many ways as possible. Many of these symbols are intended to be shown as vectorial or raster information on the terrain. In order to show these symbols, it has been implemented a whole subsystem that allows the user to modify the terrain textures. This way it is possible to represent on them all kind of raster geometry and typed text.

This far the document has only considered the rendering of grid meshes containing one texture on them. To represent real 3D symbology it is necessary to deal with a little bit more complicated rendering. The Glob3 Mobile engine uses several kinds of renderable objects to represent geospatial data, as it is shown in Figure 2:

- Meshes with a flat color.
- Meshes with a color per vertex.
- Meshes with one texture sampled.
- Multitextured meshes.
- Billboards.

- 2D HUD figures on screen space.

In addition, it’s worth mentioning that each one of them could be rendered in multiple ways, including the use of several kinds of lighting, transparencies and occlusion systems. Besides it’s remarkable that some of the characteristics of these kinds of symbols could be mutable in runtime. For instance, the user could want a billboard showing a different screen size depending of the distance or a mesh which texture mapping changes over time.

It’s now easy to see that each one of these cases should be rendered differently and provide the graphics engine specific data.

IV. GPU ARCHITECTURES

Several years ago, GPU processes run over a pipeline with stages of fixed functionality. Nowadays GPUs have dramatically change their architecture and so have newer versions of OpenGL and other graphics APIs. As there is a trend towards multi-core and general-purposed GPUs, the new idea is that, given a similar pipeline, some of the stages could be explicitly programmed by the user. [11]

In the case of OpenGL ES 2.0, these stages happen to be the Vertex Shader and the Fragment Shader, as can be seen in Figure 3. They replace many of the fixed functionality stages. The first one is intended to perform per-vertex calculations, as eye-space transformations, and the second one to perform pixel calculations once the primitives are assembled such as lighting. Each one of these functionalities will be performed in one of the GPU cores that are tuned for extremely fast floating point computation.

This new scheme allows to control exactly which are the calculations that are going to be done for each one of the drawing primitives and in addition, provide to the drawing calls the amount of data that they need whichever its nature.

The programmation of the vertex stage and the fragment stage is carried out using the OpenGL Shading Language or GLSL, that is quite similar to C. Once the GPU has the two pieces of code, it can compile and link them in what is called a Shading Program.
V. DESIGNING COMPLEX SCENES

At the first steps of the engine it was only needed to represent the terrain, which was just a set of triangle grids rendered with a texture. As all the elements of the scene were of the same nature, using a unique shading program that could render these elements seemed suitable. Many simple graphic applications are based on a single shading program which performs all the rendering. [7]

The Glob3 software architecture considers the group of elements present in the scene as a sort of tree, where the drawable elements are on the leaf nodes and their parents set common properties on them. A simple example of this is shown in Figure 4, where the root node or starting node is the leftmost one.

However, as the engine started to get more advanced new kinds of symbology were introduced and therefore, new renderable elements as can be seen in Figure 6. It was noteworthy that the amount of data passed to the GPU on each frame was growing considerably. The scene tree described before turns into a real scene graph, as a node could be accessed through many routes coming from the root node. Each one of the paths reaching a renderable node (usually a geometry node) will lead to a drawing call. As an example, Figure 5 shows a scene graph that represents two textured meshes (one of them lighted) and two flat colored meshes.

Let’s consider a scenario with two kinds of meshes using a flat color or a texture using a single program. Each time a rendering is performed, the flat color, the texture id and the texture coordinates must be set, independently of how the geometry has to be shaded. Furthermore, it has to be introduced a new variable to the shader to determine which shading to perform. This new variable will show up inside an if() clause of the shader program, which introduces a branching behaviour that affects the performance negatively. 6.

At this point, it is plain to see that the Glob3 Mobile engine should become a multi-shading software in order to keep a good performance in devices with very limited resources. The problem came from the user-oriented philosophy of the Glob3 Mobile API. Other virtual globes engines delegate the task of choosing the right shader program to the user, in a more or less explicit way[1], [4], [8]. As the background of the users is not known, it is not possible to assume that they are going to write or even choose the right shading program for a determined purpose.

The Glob3 Mobile engine should infer the shader to use and the variables that must be set, based on the structure of the scene graph that the user has created.

VI. SCENE GRAPH MANAGEMENT

Let’s consider the scene graph described in the previous section. It’s going to be composed by a great number of nodes. Each one of them is connected with the others by links that are in fact a parent-child relationship. Considering these links, the structure is a Directed Acyclic Graph where the rendering starts at the root node.

---

At the first levels of this graph are the nodes which insert in the scene the parameters shared by many elements. Lower levels contain parameters of specific portions of the scene. Finally leaf nodes represent elements to be drawn. This way there can be found nodes that represent scene parameters such as camera, lighting, texture, color or geometry.

The solution is based on the existence of a library of shading program codes. These codes could be accessed at any time by the engine to compile them on rendering time. Considering that most nodes contain necessary information for the drawing process, it is understandable that at rendering time a leaf node must know the changes introduced by upper levels. As any element can be accessed by many routes through the graph, this information is dependent of the specific route that has been followed for the drawing. In Figure 7 it is shown how a specific path through the scene graph can be represented as a chain of elements, each one containing information needed for rendering.

The chosen solution is to consider that every node owns a data structure (from now on it will be called GLState) that contains all the values that a shading program might need from the node. Each value is paired with a variable name in which it should be set.

When a certain drawable element is accessed, all the GLStates that have been visited are gathered in a single set of pairs of variables and values.

Once all the available data is gathered, it is time for resolving the conflicts that could appear in order to provide the program the right amount of information. They could belong to one of these groups:

- Incompatible data. Imagine a scenario where given a certain graph path the user have added flat color information and texture information. Considering that rendered mesh could not use both at the same time, a policy must be adopted in which one of the two sets of variables is dismissed.

- Unnecessary data. As an example consider that a geometry could provide vertex normal information. This data is generally used by lighting process but would be useless in the absence of a light source. In that case, the normals must be removed from the set of data to be passed to GPU.

Once the final set of variables to pass is achieved, the engine performs a lookup through its library of programs. If there is a program that fits the incoming data, it is chosen to be used in this drawing call. If there is not, a new program must be compiled and linked using the existing source codes.

The engine informs the graphic API that it is going to use that program. It is time for sending all the information gathered from our path in the scene graph to GPU. Afterwards, a draw order will be sent to OpenGL so it performs the rendering.

VII. PERFORMANCE UPGRADES AND IMPLEMENTATION CONSIDERATIONS

It is to be noticed that the quite long process described above is going to be run before each draw call sent to OpenGL. This means, thousands of times for each frame. Several implementation issues should be taken into account for maintaining a suitable frame rate.

A. Scene graph structure

It is noteworthy that many of the final applications ended in a quite immutable scene graph. Furthermore, most of the nodes were only accessed by one node all over their lives. That is specially true for top level nodes. The engine will try to render these most common cases in the quickest way possible, despite of the fact that any kind of mutable scene graphs are supported.

Taking that into consideration it was proposed an implementation where every GLState contains a reference to its parent GLState and a timestamp which monitors any modification performed on itself. Any change on the structure of the state or on its parent is reflected on a new timestamp. This way, if a GLState is accessed for a second time by the same parent with the same timestamp, it’s sure that the whole chain of ancestors remains the same. Thus, each GLState can maintain a resume of all the information gathered in the last path and reuse it on every frame rendering as long as it does not change. This resume will be stored in the last node of the chain as shown in Figure 8.

This way, whenever a GLState belonging to a drawable node is reached, it already has a shortcut to all the variables that must be set, avoiding the gathering process and the conflict resolution.
B. Shading programs look up

Once the set of data that the shading process is going to need is determined, the engine must perform a search task among the shading program candidates. As previously said, the path to each renderable node will be static in most of the cases. This implies that the information contained in the path will be the same and, therefore, the shading program lookup will return the same program.

Considering this, it is much more efficient to keep a reference to the program that was used in the last frame and reuse it the next time the node needs to be drawn. This reference will be dismissed any time the node is rendered through a new path.

In a completely static scenario, programs lookup would only occur at the first frame.

C. Render information setting

From an OpenGL API perspective, the rendering process consists of sending data to GPU (vertex data, textures, indices...), setting a specific state for the drawing (enable depth test, set which texture to use...) and commanding the drawing operations. These steps are the same for every element within the scene.

It’s remarkable that the two first steps can be way more time consuming than the drawing process itself. Besides, in scenes like the virtual globe, where many elements can be drawn repeatedly with minimum variations, many of the API calls to set the OpenGL status could be redundant.

In order to minimize these redundant calls, the shading programs retain a copy of every value that is set to their variables. This way, anytime a value is set for a second time in a shader variable, a simple comparison prevents the engine to perform an API call and a data transfer to GPU memory.

Experiments show that for a simple symbology example up to 80% of set calls can be avoided using this technique. The performance will be even better if state sorting techniques are used [6].

VIII. CONCLUSION

Programmable GPU technology has open a wide range of opportunities ahead of us. For instance, we have started working on implementing new solutions for some of the problems implicit on the rendering of huge and complex scenarios using General Purpose GPU techniques.

Another promising step deserving attention is the automated generation of source codes for new shading programs at runtime. This will allow the Glob3 Mobile users to draw using a broader range of programs while developers will not need to explicitly create them on GLSL.

ACKNOWLEDGMENT

The third author wants to thank Agencia Canaria de Investigación, Innovación y Sociedad de la Información, for the grant “Formación del Personal Investigador-2012 de Gobierno de Canarias”.

REFERENCES


