Agreste Game Engine: Development of a Dedicated OpenGL Game Engine Proposal

Joimar Brito Gonçalves Filho, João Gabriel Lima Moraes, Victor Travassos Sarinho
Universidade Estadual de Feira de Santana (UEFS)
Laboratório de Entretenimento Digital Aplicado (LEnDA)
Feira de Santana, Bahia, Brazil
joimarbgf@gmail.com, joaofeirense@gmail.com, vsarinho@uefs.br

***Resumo*—OpenGL is an important graphical development API for digital game, but with a low level of public documentation and tutorial examples explaining how to use it. In this sense, the present work describes the attempt to develop a game engine based on this API. The Agreste Game Engine (AGE) aims to understand the concepts involved in the manipulation of OpenGL resources, which is not directly specified to provide new game engines. As obtained results, interactive virtual objects were provided by AGE prototype, showing the AGE capability of using OpenGL resources according to developed graphics and physics engines in a dedicated platform, as well as a public example about how to use OpenGL in the production of new game engines.**

***Keywords*-OpenGL; game engine tutorial; dedicated game engine; agreste game engine;**

I. Introduction

The use of OpenGL for the development of electronic games has been widely diffused in the XXI century [1]. It is an API that directly handles the computer graphics processing unit (GPU), thus enabling the optimization of 3D graphics involved in the production of digital games and game engines [2].

Per game engines, they allowed the automation of the gaming industry, which had a previous production scenario of almost craft production [3]. It is a technology that arose from the need to avoid redundant work in different games with similar aspects, providing an extensible software that can be used as the foundation for many different games without major modification [4].

Overall, exploring OpenGL features directly is an arduous task that is often avoided by the digital game developers community, becoming popular game engines that encapsulate such features [4]. This is an activity that requires GPU handling in low-level languages, yet embodying high-level programming paradigms and maintaining graphical rendering performance [5].

This paper describes the development and lessons learned of *Agreste Game Engine* (AGE), a 3D game engine based on the direct manipulation of the OpenGL API for a dedicated platform. It seeks to understand the concepts involved in the direct manipulation of this API, providing as a result graphics and physics engines together with the OpenGL knowledge about how to use it in the production of dedicated game engines.

To this end, section 2 describes relevant theoretical aspects for the AGE production. Section 3 presents the design, implementation and prototype results for the proposed game engine. Finally, section 4 describes the conclusions and future work of this project.

II. Theoretical Aspects

A. Graphics Engine

The graphics engine is a program responsible for receiving input from the developer and transforming it into code that can be interpreted by OpenGL for creating and updating frames [4]. The input can be defined in: models (3D or 2D), textures, animation loops, light effects, camera positioning, text, or information that can be extracted from input and output devices.

From this, OpenGL becomes responsible for managing operations necessary for the production of a frame, which, according to the pipeline [6], should be sent to the GPU through a framebuffer.

1) Shaders: Shaders play an important role in the process of rasterizing a frame. The rasterizer, an entity responsible for delimiting the paint zones of a frame using geometric data, uses the shaders to paint the described entities with their respective color pattern. Shaders are programmed to print colors in a specific way to the area in which they are used [6].

2) Vectors: Mathematical entity used in analytical geometry to describe a point in a dimensional space, endowed with direction and direction module. In computer graphics, vectors are used to model entities in a given 2D or 3D environment [6].

3) Textures: A texture (also known as texture map) is basically an array of colors, either loaded from a file (such as a jpeg or targa file), or occasionally generated by procedures. Usually, these arrays have two dimensions, but there are 1D (line with different colors) and 3D (3D texture was considered as a stack of 2D textures). 3D textures are commonly used in medical imaging, while a 1D texture can be used to store the result of a rendering function such as a sine wave or a linear interpolation factor. A texture will usually have 8 bits per pixel, with four color channels. But there are specific context variations with 16 bits per pixel, and images with fewer channels, such as black-and-white

maps or completely opaque images, which do not make use of the alpha. Unlike an ordinary image on a screen, the elements of a texture are not called pixels, but textels, short for texture element [6].

4) Matrix Transformations: In computer graphics, matrix transformations are extensively used, allowing any combination of rotation, translation and scale operation in a vector. There are 2 types of matrices used: template matrix and display matrix [6].

5) Depth: The depth is displayed on the screen from the overlapping entities that make up a frame. Each entity printed on screen has a Z value assigned to its pixels. This value determines in which layer the portion of the image will be printed, the so-called depth of the image. For example, if two objects overlap each other, the Z value will determine which entity stored in the depth buffer should be displayed superimposed on another, giving the impression that one object is in front of the other [6].

B. Physics Engine

The physics engine is a program designed to handle all kinds of events involving simulation of physical phenomena within a game, giving functions that deal with general physics behaviors. However, since a versatile game engine should give diversified options to the developer, it is not interesting to realistically treat the phenomena that occur in a game [7].

All entities in the virtual environment can be mapped through their position in vector space. As these entities can be abstracted into a set of vectors, they allow the matrix transformation equations usage to simulate physical phenomena applied to them [7]. To do this, the following kinematics functions [7] were used:

$V = V_0 + at$ (1)

$V_{x+1} = V_x + a_x t$ (2)

$S_{x+1} = S_x + V_{x+1}\Delta t$ (3)

Knowing that acceleration and velocity are vectorial quantities, it is possible to determine the final position of any vector that follows this displacement pattern.

C. The Rasterization Pipeline

All objects to be drawn in a scene must be described at the input of the pipeline through geometric data (the vertices that make up the image). In this sense, necessary geometric modifications are made on the inserted vertices, such as: the spatial specification point by point; geometric transformations (rotation, translation and scale); cropping, if the image goes beyond the specified print area in the frame; perspective division, that defines the size relative to the spectator view for each object printed on screen; and camera transformation, which defines the angle under which objects are observed. After perform these operations, it is time to paint the pixels described in the image, called *rasterization process*. In addition, an algorithm is applied that defines the priority of printing the pixels, giving the idea of image depth and the alpha application that defines the transparency of a pixel. All of this is written in the memory space of the graphics card called FrameBuffer.

III. The Agreste Game Engine

A. OpenGL Resources

Some OpenGL libraries were required by the AGE development environment (Visual Studio 2015) to provide an initial version of the proposed game engine:

- **GLM** - Mathematical library oriented to the C++ language and based on OpenGL Shading Language (GLSL) that contains functions that abstracts the calculation of matrix operations, data storage and noise [8].
- **GLFW** - Simple API for creating windows, contexts and surfaces, receiving input and events.
- **GLEW** - Tests the compatibility of the version of the installed OpenGL with other libraries installed and with the operating system used [9].
- **SOIL** - Library that allows you to import textures to an application without requiring conversion to bitmap or own image formats such as the GLFW library.

An initial study was also performed to delimit the scope of the graphics engine, defining as a result how OpenGL performs the rendering process and how the functions in its rasterization pipeline are delegated during this process. As a result, it was noticed that OpenGL functions specifically manages simple geometric models such as points, lines, triangles, squares and circles for the production of more complex entities, as described below:

- **linestrip** - A structure used basically for the representation of graphs and paths assembled through straight segments, determined by the vectors delimiting each segment.
- **lineloop** - Like linestrip, lineloop uses a similar algorithm, however, its objective is to render a closed graph, joining the first to the last vertex described, and defining a geometric form as result.
- **trianglestrip** - As well as straight lines, where vertices taken 2 to 2, OpenGL can perform a similar process using vertices taken 3 to 3, reusing the two vertices of the last triangle described and riding a geometric figure composed of agglutinated triangles.
- **trianglefan** - Follows the logic of assembling geometric forms from more primitive entities, through the association of triangles from the same axis of rotation, thus being able to form more complex figures, like circles, for example.
- **squarestrip**- Similar to the triangle strip, squarestrip performs a process of agglutination of vertices taken four to four to form a geometric entity, reusing two of the vertices of the last quadrilateral described to define a new polygon.

B. AGE Design

Considering the AGE architecture (Figure 1), it is based on the interaction of the graphics and physics game engine resources. The graphics engine interacts with the GLFW library [10], to create windows that will receive plotted frames. This library is also responsible for receiving user inputs from devices like gamepad, keyboard and mouse. The gathered input is interpreted as mathematical data, which is used to calculate new positions in the Model Matrix, by the physics engine.

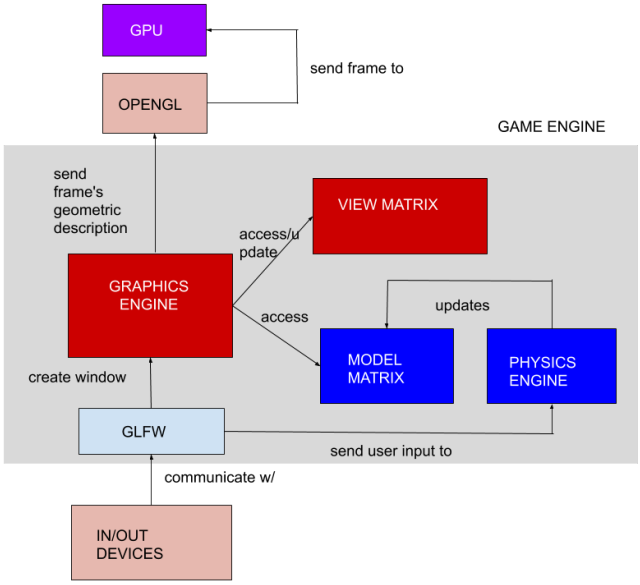

Figura 1. AGE system architecture.

The Model Matrix is accessed by the graphics engine in order to update the view matrix. This matrix is used to perform various operations like clipping, camera rotation and perspective. After completing operations, the graphics engine can send the frame data to OpenGL, that uses shading technologies, such as the OpenGL specific shader description language (GLSL), to manipulate the GPU and to create the images to be shown at the screen.

Regarding the physics engine, kinematics equations were applied on created models, assigning them mass and speed. Each of these components were analyzed by interacting with the geometric transformation operations. For this purpose, a number of transformation matrices were developed according to the desired operation.

In the system, the manipulation of physics engine is still indirect. Through manipulation of gravity is possible to see the effects of the physics engine on the created objects. All the objects are equally under the effect of gravity, The position of each element on the frame is calculated and updated in real time by using the kinematic equations (1), (2) and (3).

C. AGE Implementation

AGE works by compiling user codes for the created objects and, at the same time, presetting necessary entities for the games like light and camera, which are called basic objects. The user might change the values set for the basic objects in the game by modifying their attributes. For instance, the user might be able to: change the colour of a light source by using a color picker; configure a hexadecimal value directly in the code; change the angle of the camera by dragging and dropping the mouse cursor; or inputting the position values in a specific field. Figure 2 illustrates one possible configuration of one game object, in this case a light with predefined color and direction values.

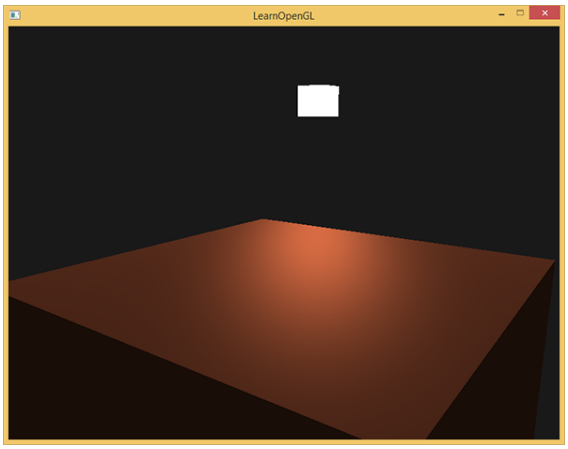

Figura 2. Configured game light object with predefined color and direction values.

Another implemented resource for the graphic engine was the import system of 3D models — the main source projects created in software Blender. The process of importing content from third party software consists of gathering all mathematical data from the object, in order to calculate the position of the pixels. The rasterization occurs by interpreting the pixels as primitive forms such as proposed by OpenGL functions.

As for the physics engine, the module is static, applying the gravity force to all rendered objects, and it uses the GLM library to calculate all necessary vectorial data to be correctly rendered by the graphic engine.

D. AGE Prototype

The AGE proposal is to build 3D platform games, and it was designed to be or a game composed of many scenarios and objects or a huge connected world, each one with their characteristics and limitations. In fact, work with different scenarios has the drawback of swapping between maps which can greatly decrease the performance. In the same way, work with a single huge map might present the disadvantage of the program having to load many elements at once.

In this sense, and considering the initial AGE development results for 3D platform games, it is possible to import geometric models and manipulate them in order to give movement and mass, according to user configuration values. It is also possible to assign colors to the vertices of an object, according to their respective vector positions and interpolate them. As an example, Figure 3 illustrates a rendered cube by the AGE prototype, using the color interpolation shading technique.

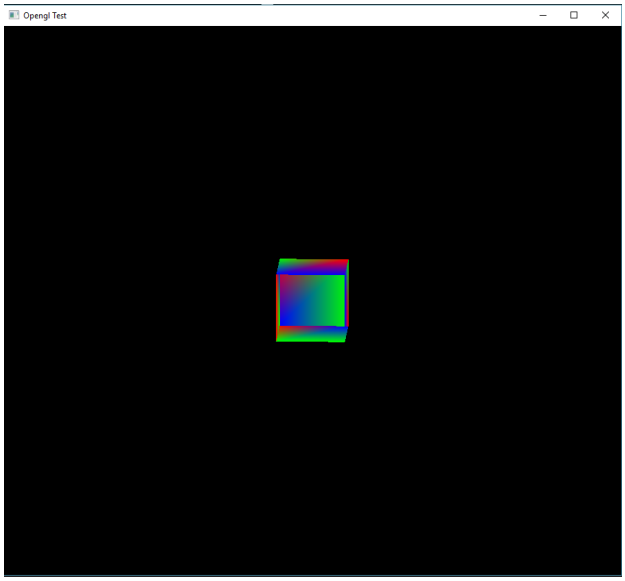

Figura 3. AGE rendering coloured cube.

IV. Conclusions and Future Work

The objective of demonstrating the graphic power of OpenGL library manipulation by the AGE in a dedicated platform was achieved. For this, the encapsulation of the OpenGL functions for the development of the graphic engine was performed, as well as the use of GLM functions to perform the necessary calculations that the physics mechanism applies in the Model Matrix.

The developed graphic engine is able to make essential graphics operations used by many types of game engine (in special for 3D platform games), such as vector manipulation, colour interpolation, making 3D solids and customized textures assimilation. The physics engine is also able to simulate phenomena such as gravity and dumping factor. The interactions with user commands also work successfully, being able to use keyboard, mouse or gamepad interfaces to interact with rendered objects.

Regarding the AGE development challenge, despite the wide OpenGL documentation, it is not directly specified to the production of game engines. On this matter, an exploration work was performed, often manipulating libraries that are shortly documented. In this sense, for educational and academic purposes, the AGE development represents an important effort to provide an open tutorial example about how to develop game engines with OpenGL. Moreover, the AGE creators team will also release a documentation guide in the future with instructions about how to use OpenGL and support libraries (previously quoted) to make games and game engines.

Also as future work, AGE needs to be upgraded since must of the implemented functionalities should be encapsulated within user friendly functions, to be used according to the developer needs. The inclusion of important production resources available in professional game engines, such as light maps, collision detection and fluid mechanics, will be also attached to AGE in the future. Finally, for evolution purposes, the AGE migration to the Vulkan [11], a low-overhead, cross-platform 3D graphics and computing API that is listed to replace OpenGL, will also be performed in the future.

Referências

[1] *OpenGL Overview*, https://www.opengl.org/about/, Acessed in 01/07/2019, mar. de 1997-2019.

[2] Jeremiah, *Introduction to OpenGL for Game Programmers*, https://www.3dgep.com/introduction-opengl/, Acessed in 01/07/2019, fev. de 2011.

[3] H. Lowood, "Game Engines and Game History", *Kinephanos*, n.º 2, pp. 179–197, jan. de 2014.

[4] J. J. Michael Lewis, "Game Engines In Scientific Research", *Communications of the ACM*, pp. 27–31, jan. de 2002.

[5] A. Asaduzzaman e H. Y. Lee, "GPU Computing to Improve Game Engine Performance", *Journal of Engineering and Technological Sciences*, vol. 46, pp. 226–243, jul. de 2014. DOI: 10.5614/j.eng.technol.sci.2014.46.2.8.

[6] D. G. Ushaw, *Notas da disciplina graphics for games, csc3223.NewcastleUniversity*, Newcastle University, jan. de 2014.

[7] D. W. Blewitt, *Notas da disciplina gaming simulations, csc3222.Newcastle Universiy*, Newcastle University, jan. de 2015.

[8] C. Riccio, *OpenGL Mathematics - A c++ mathematics library for graphics programing*, https://glm.g-truc.net/0.9.8/index.html, Acessed in 01/07/2017, out. de 2002-2016.

[9] N. Stewart, *The OpenGL Wrangler Library*, http://glew.sourceforge.net/, Acessed em 01/07/2017, out. de 2017.

[10] M. Geelnard, *GLFW - An OpenGL Library.* http://www.glfw.org/documentation.html, Acessed in: 01/07/2017, out. de 2006-2011.

[11] G. Sellers e J. Kessenich, *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016.