A Shader-Based Architecture for Virtual Reality Applications on Mobile Devices

Adriano M. Gil(✉) and Thiago S. Figueira(✉)

SIDIA Instituto de Ciência e Tecnologia, Manaus, Brazil
{adriano.gil,thiago.figueira}@sidia.com

Abstract. As new technologies for CPUs and GPUs are released, games showcase improved graphics, physics simulations, and responsiveness. For limited form-factors such as virtual reality head-mounted displays though, it is possible to explore alternatives components to harness additional performance such as the GPU. This paper introduces a shader-based architecture for developing games using shared resources between the CPU and the GPU.

1 Introduction

Virtual reality (VR) brings the promise of a revolution in the way entertainment is consumed in present times. The user is placed at the center of the action and perceives content from every direction.

Games, for their part, transport players to a world envisioned by game designers and developers. As the technology for other form factors such as PC and console advances, VR players want life-like graphics and improved responsiveness.

Modern mainstream game consoles and PC sets allow parallel computing to be performed. In order to harness this extra computing power, it is necessary to move tasks from the single threaded game loop and place the ones that can run in parallel in different processors.

VR devices, on the other hand, have a limited form factor. Issues like the heat generated by the processing components have to be taken into account which means adding more computing power is not possible without having side effects for the final user through the current form factor.

In VR games development, the most used game engine is *Unity* [12] and even though it is optimized, we believe there is an opportunity in exploring graphics cards for additional performance.

In this work, we propose an implementation of the classic game Snake using a shader-based architecture. By using a logic based of parallel execution we achieved a performatic virtual reality application in which every visual element is defined and rendered by the shader in a unique mesh.

Related research is explored in Sect. 2. An explanation about the game is provided in Sect. 3. Important concepts for this topic, Game Loop and Game Architecture are explored in Sects. 4 and 5. The proposed architecture is explained in

Supported by SIDIA.

C. Stephanidis et al. (Eds.): HCII 2020, LNCS 12428, pp. 101–111, 2020.
https://doi.org/10.1007/978-3-030-59990-4_9

Sect. 5.1. Section 6 describes how the game logic is managed. Section 7 analyzes how shaders are used for rendering the game. At the end, results are in Sect. 8.

2 Related Work

The literature is scarce about approaches using GPU development for Virtual Reality though there are some initiatives in mobile and computer environments.

The two-dimensional game *GPGPUWars* [5] has its code structure based on *shaders*, similar to the architecture presented here where the GPU performs all the processing of the game. The mobile game MobileWars is a massive 2D shooter with top-down perspective [6] which uses the GPU to process the game logic and the CPU for the data acquisition step. However, applications in virtual reality differ from other applications because there is the need to fill the three-dimensional space to provide content for 3 degrees of freedom (3DoF-*3 Degrees of Freedom*). For example, the application described in [15] uses computational vision to generate a panoramic view of an 8-bit console game.

Some works explore different game architectures: AlienQuiz Invaders [4], for example, is an augmented reality game that implements cloud services to improve overall game quality. [5–7] explore the GPU for performance in mobile and PC environments.

3 The Game - VRSnake

VRSnake is the virtual reality version of the classic 2D game Snake. In the original game, the player moves the snake to collect the elements that appear randomly during gameplay. In its virtual reality adaptation, VRSnake, the player decides where to position the collectable items rather than directly controlling the snake. In other words, the player stands in the center of the game world surrounded by the inverted sphere where the snake moves. For this new gameplay design, we defined the following rules:

1. The player sets the collectible object position;
2. The snake continuously and automatically seeks the collectable object placed by the player and grows in a unit when it reaches this object;
3. The victory condition is making the snake hit itself during the pursuit for the object.

4 Game Loops

The game loop is the foundation which upon games are built. Games are considered real-time applications because their tasks rely on time constraints. According to [7], these tasks can be arranged into three steps: data acquisition, data processing and presentation. The first step is about collecting input data from the input devices (for VR devices, it is the Head-Mounted Display (HMD) and

the joysticks); the second step is applying player input into the game as well as game rules and other simulation tasks; the last step is providing to the user the current game state through visual and audio updates.

There are two main groups of loop models proposed by [13]: the coupled model and the uncoupled model. In the simplest approach, all steps are executed sequentially and it runs as fast as the machine is capable of. The uncoupled model separates rendering and update steps in different threads, but this may cause the same unpredictable scenario of the Coupled Model when executed in different machines. The Multi-thread Uncoupled Model feeds the update stage with a time parameter to adjust its execution with time and allow the game to behave in the same way in different machines.

Due to its interactive nature, these steps should be performed as fast as possible in games or performance may jeopardize user experience. For VR applications, this constraint is even heavier as VR games should run at 60 frames per second [11] to avoid nausea and other negative user effects, this is why Virtual Reality software requires powerful CPU and GPU hardware [2].

Modern mainstream game consoles and PC sets allow parallel computing to be performed. In order to harness this extra computing power, it is necessary to move tasks from the single threaded game loop and place the ones that can run in parallel in different processors (Figs. 1 and 2).

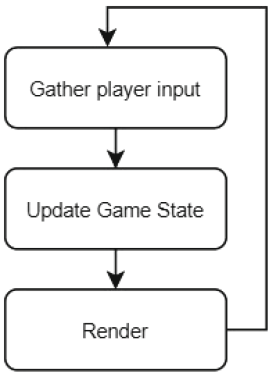

Fig. 1. Coupled model

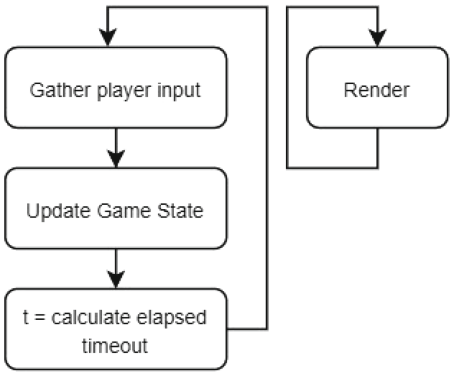

Fig. 2. Multi-thread uncoupled model [13]

5 Game Architectures

Games are a software product, therefore the architecture of a game is comparable to that of software and defines how the game is built. Usually, it is not apparent to the player, except for performance [1].

According to [8], it is possible to classify any sub-system in a game in one of the three categories (see Fig. 3): the application layer, the logic layer, and the

game view layer. The first layer deals with the operating system and hardware. The second layer manages the game state and how it changes over time. The game view layer presents the current game state with the graphics and sound outputs.

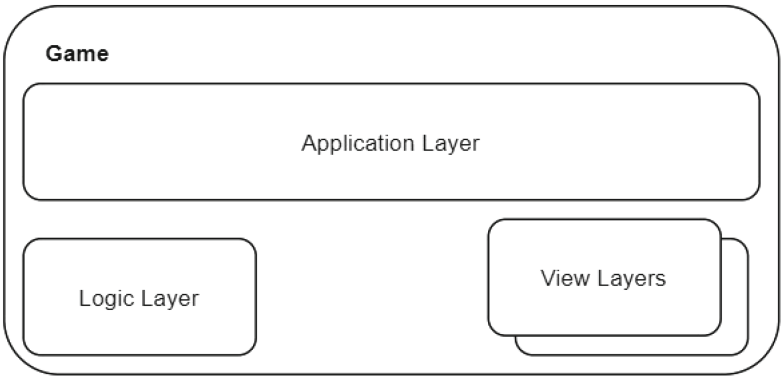

Fig. 3. High-level game architecture

Game engines provide *"software that is extensible and can be used as the foundation for many different games without major modification"* [2], and as such, they encapsulate the general game architecture (see Fig. 4).

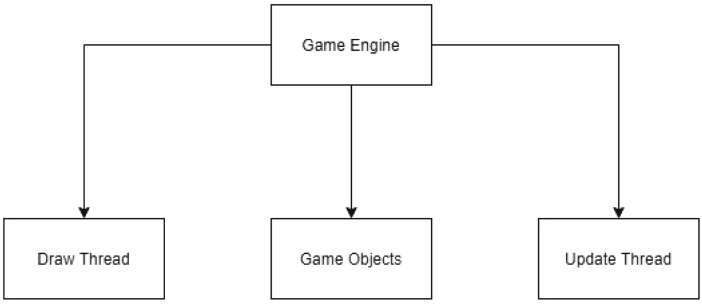

Fig. 4. Simplified view of a game engine architecture

The game engine is responsible for the Update Thread, the Draw Thread and the Game Objects. Each correspond to a general step defined in the general game architecture. The Game Objects are the items the player may or may not interact with during game-play. The Update Thread is responsible for updating game objects as far as it can whereas the Draw Thread updates all elements visually on the output device [10].

Game architectures can be updated to better reflect the needs of the game. A well-defined architecture enables better performance and overall results with the final game. In fact, most game engines (as well as the architectures behind it) are crafted for specific platforms and, in some cases, for particular games [2].

5.1 Proposed Architecture

We propose a game architecture based on two layers: one layer handles the game logic whereas the other manages rendering. The logical layer is CPU-bound and has tasks such as the search for the collectible objects and snake movement.

The visualization layer involves a shader, a piece of code that runs in the Graphics Processing Unit (GPU), that renders all game objects on the output device, which includes the collectible objects as well as the snake.

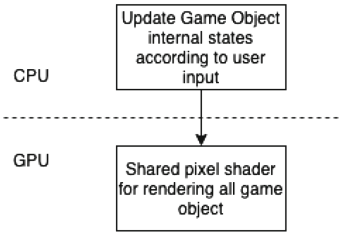

Fig. 5. Logic layer runs on CPU

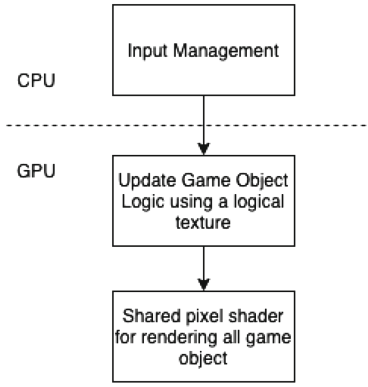

Fig. 6. Both logic and visualization layers runs on GPU

In other words, the logical layer manages the collision and movement of the snake and selects the most promising path given a randomization factor; The visualization layer renders all the elements arranged in the output device, the CPU does not influence these objects.

Our implementation is based on pixel shaders since we want to guarantee support for Android-based VR platform that don't have support for more specialized shader types, like computer shaders for instance.

Figures 5 and 6 show diagrams of how our proposed architecture can be devised considering CPU and GPU-bounded scenarios. For mitigating heavy computation on the GPU, a logic layer on CPU is recommended, in the other hand the intensive usage of multiple elements on the logic layer can indicate the GPU as the best solution.

6 Logic Layer

The logic layer is responsible for handling user input and its correspondent actions inside the game virtual word. Thus it is necessary to model how the game works. In VRSnake, the snake modeling includes how it moves and how it interacts with each game element.

In VRSnake, the user input is based on an external joystick. As a sensor, however, these controllers are subject to noisy interference during the virtualization of the event represented in the real world by the user's movement. In order to improve the signal capture and translate the player's intentions more faithfully, the Kalman filter [14] is applied to the readings of the joystick events.

6.1 Managing Game Objects

Our framework is based on managing objects data that are rendered by pixel shaders. Thus we made use of textures to store logic data, that we named as logical textures. The visualization layer should localize the right position of the data inside the logical texture to render the content. One of the disadvantages of such method is more expensive to spread big changes (Fig. 7).

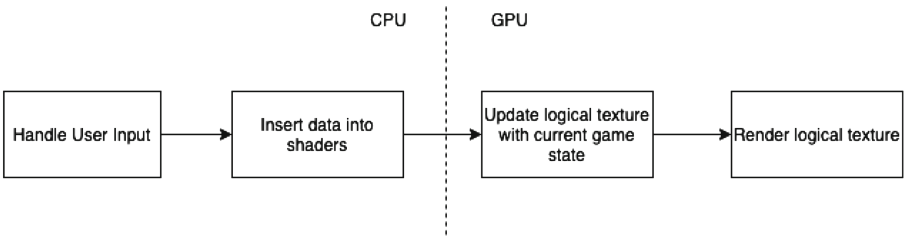

Fig. 7. Our proposed data management approach for game elements

The logical texture is generated by blit operations, i.e., copying the output of a pixel shader to a texture buffer. Its final size is defined by overall size of the game world. As we designed a 50×50 grid, it's required a 50×50 texture to store all the information regarding the game elements. Thus an identification value for each game element is stored in the right position of the grid. After each game update, the next blit call should update the current snake position. By looking ahead of the movement direction in a 3×3 neighborhood, it's possible to know whether a snake block will move.

6.2 Snake Movement Agent

The movement of the snake is managed by a state evaluation function that analyzes each possible action at any given time. In essence, the serpent is always seeking the collectible objects, so it evaluates the shortest distance course in the

X and Y axes in UV space and, provided that there is no possibility of hitting itself, proceeds through this path and repeats the process. The function below illustrates this procedure:

$$F(A) = R * (D + O) \tag{1}$$

Where R is a randomization factor; D represents the *Manhattan* distance between the current position and the collectable object; And O is a value attributed to the existence or not of obstacles in this path.

7 Visualization Layer

The visualization layer has the code necessary to make game elements visible to the user. In the proposed architecture, the visualization layer is based entirely on a pixel shader running in a single mesh.

Given a logical texture, i.e. a texture representing the current game state, the centralized pixel shader must render the position of each game element: snakes and food. The role of the visualization layer is to represent the game element according to a style: 2D or 3D for instance. In this work, we implemented a 2D representation on a sphere and a raymarched visualization on a cube. However many others are possible, like using particles for example.

The following subsections explain how snakes in VRSnake are drawn. Subsect. 7.1 explains the general method, whereas the others explore the specifics such as inverted spheres (Subsect. 7.2), 2D shader snakes (Subsect. 7.3) and raymarched snakes (Subsect. 7.4).

7.1 Rendering VR Snakes

Pixel shaders are scripts that carry the mathematical calculations and algorithms to compute the color of each pixel rendered on the output device. For 2D and 3D snake rendering, the shader in the visualization layer requires an external texture that contains the logical information needed to draw the final image.

7.2 Virtual Reality in a Inverted Sphere

The immersion sense that comes with virtual worlds requires visual information available from all angles. Given that our proposition considers a 2D game, the challenge is displaying two-dimensional content in a 3D scenario with the user at the center.

An inverted sphere, a sphere that has only its inner side rendered, makes it possible to fill the entire field of view, it also is the endorsed solution to display equirectangular images in 360°. The procedural generation of a sphere can follow one of the two approaches below:

1. An icosphere, i.e., a sphere which vertices are evenly distributed;
2. Generation of vertices based on longitude/latitude coordinates.

For this work, the second approach was adopted due to the possibility of using longitude/latitude as a way to map the UV coordinates through the equation below:

$$R^2 \leftarrow R^3 : (\lambda, \theta) \rightarrow (x, y, z) \tag{2}$$

To calculate the positions of the sphere vertices, given $N_{latitude}$ latitude values and $N_{longitude}$ longitude values, the value $R_{longitude}$ is defined as the angular longitude size of a cross-section of the sphere, as seen in the Eq. 3.

$$R_{longitude} = \frac{2\pi}{N_{longitude}} \tag{3}$$

The total angular size of an amount of i of longitude values can be given by the Eq. 4.

$$\alpha_i = i * R_{longitude} \tag{4}$$

Equations 5 and 6 define the X and Z positions of sphere points belonging to a cross section of the sphere that has radius D.

$$x_i = D * \sin(\alpha_i) \tag{5}$$

$$z_i = D * \cos(\alpha_i) \tag{6}$$

In a longitudinal cut, it is possible to notice that the radius D of the cross section is variable along the height of the sphere. It is then determined a value R as the angular size of a latitude value of the sphere, as seen in the Eq. 7.

$$R_{latitude} = \frac{\pi}{N_{latitude}} \tag{7}$$

The total angular size of an amount of i of latitude values can be given by the Eq. 8.

$$\alpha_{latitude} = i * R_{latitude} \tag{8}$$

The Y position of the sphere points, considering unit radius, can be given by the Eq. 9.

$$y_i = \cos(\alpha_{latitude}) \tag{9}$$

The radius D_{yi} obtained in a cross section at latitude i is defined in the Eq. 10 as:

$$D_{yi} = 2 * \sin(\alpha_{yi}) \tag{10}$$

By applying the Eq. 10 in the Eqs. 5 and refequation4 the X and Z positions of the sphere vertices are obtained according to their longitude and latitude coordinates.

$$x_i = 2 * \sin(\alpha_{latitude}) * \sin(\alpha_{longitude}) \tag{11}$$

$$z_i = 2 * \sin(\alpha_{latitude}) * \cos(\alpha_{longitude}) \tag{12}$$

Equations 10, 11 and 12 allow to define a vector P_i with components (x_i, y_i, z_i) for each uv position. By using all those equations we can find all the vertice positions of a procedural sphere.

7.3 Simple 2D Snakes

In VRSnake, the shader responsible to render the snake requires an external texture that defines the current game state. This texture as well as other game elements such as snake color and game background can be defined in the user inspector in Unity (see Fig. 8).

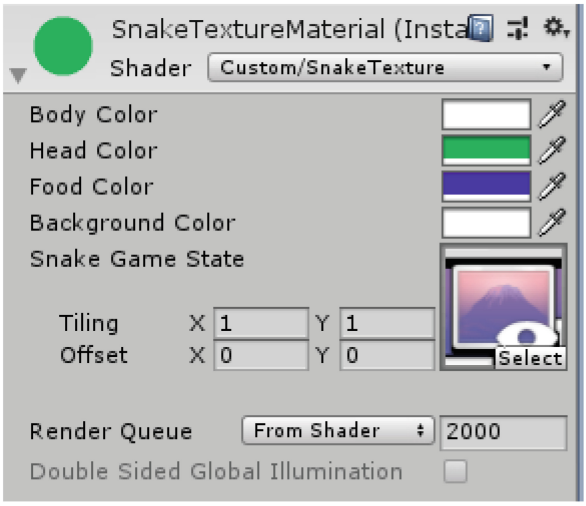

Fig. 8. Snake rendering shader in Unity Editor

The external texture is managed by the logic layer, which updates the snake during the update loop of the game. The shader layer sweeps this texture and gathers the red parameter of the RGB color system. The shader reads those values and converts each pixel to screen information according to arbitrary value ranges as seen below:

- Between 0–0.25 - Draws the head of the snake
- Between 0.26–0.5 - Draws the body of the snake
- Between 0.51–0.75 - Draws the collectible object
- Between 0.76–1 - Draws the background

7.4 Raymarching Cube-Based 3D Snakes

Raymarching is an iterative approach for rendering a scene. In each iteration, a ray moves through a line towards a fixed direction and only stops when it reaches an object. Differently from raytracing, it is not necessary to calculate the intersection point between a line and a geometric model. It should be calculated only if the current point is in the geometric model.

Using signed distance function (SDF) techniques [3], it is possible to find the distance from the current point position until the target geometry. By means of SDFs, it is possible to accelerate the iterations of raymarching by jumping straight to right position of the intersected geometry.

Our proposal can also be implemented as fake 3D objects rendered by pixel shaders in a plane mesh. We employed raymarching to render cube-based 3D snakes inside a real cube.

8 Experiments and Results

The Snake game was developed for virtual reality in the Unity engine. Figure 9 illustrates the frame rate in the GearVR through the Oculus performance assessment tool, the *OVR Metrics Tool* [9]:

The GearVR is the virtual reality headset from Samsung which is designed to interact with smartphones. In partnership with Oculus, it is possible to download VR apps and games from the Oculus store.

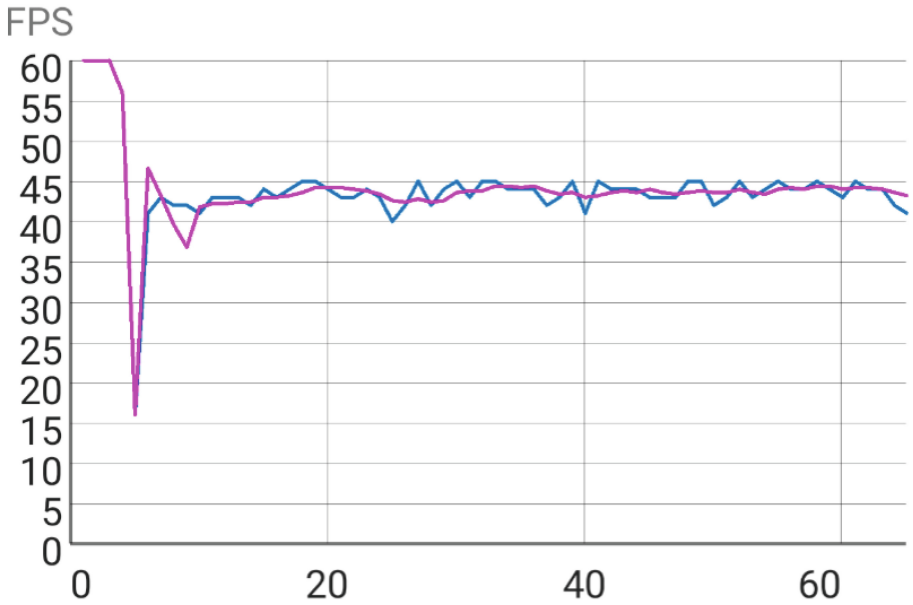

Fig. 9. Frames-per-second recored by the OVR Metrics Tool

The application presented an average frame-rate of 43.96 fps (frames-per-second), with a minimum 16 fps and maximum of 60 fps. As represented by the Fig. 9, the application does not 60 fps and this is most likely due to the kalman filter implementation in the data collection step of the game loop and the garbage collector.

9 Conclusions

We believe it is possible to explore the GPU for additional performance of virtual reality games in mobile devices. The architecture defined here can be explored as an additional tool for developers to complement the way the game looks and manage resources of VR applications. For future work, we see an opportunity to bring the steps currently present in the logic layer to the visualization layer, further exploring the GPU. We also envision bringing machine learning techniques to the visualization layer to allow multiple snakes to share the same space thus increasing game difficulty. There is the possibility of further improving performance through optimizations to the input gathering implementation as well.

References

1. Croft, D.W.: Advanced Java Game Programming. Apress, New York (2004)
2. Gregory, J.: Game Engine Architecture, 3rd edn. CRC Press, Taylor & Francis Group, Boca Raton (2019)
3. Hart, J.C.: Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. Vis. Comput. **12**(10), 527–545 (1996)
4. Joselli, M., et al.: An architecture for mobile games with cloud computing module. In: XI Brazilian Symposium on Computer Games and Digital Entertainment. SBGames 2012 - Computing Track (2012)
5. Joselli, M., Clua, E.: GpuWars: design and implementation of a GPGPU game. In: 2009 VIII Brazilian Symposium on Games and Digital Entertainment, pp. 132–140. IEEE (2009)
6. Joselli, M., Silva, J.R., Clua, E., Soluri, E.: MobileWars: a mobile GPGPU game. In: Anacleto, J.C., Clua, E.W.G., da Silva, F.S.C., Fels, S., Yang, H.S. (eds.) ICEC 2013. LNCS, vol. 8215, pp. 75–86. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41106-9_9
7. Joselli, M., et al.: A game loop architecture with automatic distribution of tasks and load balancing between processors. In: Proceedings of the VIII Brazilian Symposium on Computer Games and Digital Entertainment, January 2009
8. McShaffry, M.: Game Coding Complete, 3rd edn. Course Technology PTR, Boston (2009)
9. OVR Metrics Tool. https://developer.oculus.com/downloads/package/ovr-metrics-tool/
10. Portales, R.: Mastering android game development. https://www.packtpub.com/game-development/mastering-android-game-development
11. Pruett, C.: Squeezing performance out of your unity gear VR game, May 2015. https://developer.oculus.com/blog/squeezing-performance-out-of-your-unity-gear-vr-game/
12. Unity for all (2019). https://unity.com/
13. Valente, L., Conci, A., Feijo, B.: Real time game loop models for single-player computer games (2005)
14. Welch, G., Bishop, G.: An Introduction to the Kalman Filter. University of North Carolina, Department of Computer Science (1995)
15. Zünd, F., et al.: Unfolding the 8-bit era. In: Proceedings of the 12th European Conference on Visual Media Production, p. 9. ACM (2015)