Journal of Computer Science

Original Research Paper

Graphics Pipeline Evolution: Problems and Solutions

[1]Ghazi Shakah, [1]Mutasem Alkhasawneh, [2]Victor Krasnoproshin and [3]Dzmitry Mazouka

[1]*Department Computer Sciences, Faculty of Information Technology, Aloun National University, Jordan 26810, Ajloun*
[2]*Department of Information Management Systems Faculty of Applied Mathematics and Computer Sciences, Belarusian State University, Belarus*
[3]*New Zealand, 2/26 Sunset Road, Unsworth Heights, 0632 Auckland, New Zealand*

Article history
Received: 11-12-2018
Revised: 27-03-2019
Accepted: 01-07-2019

Corresponding Author:
Ghazi Shakah
Department Computer Sciences, Faculty of Information Technology, Aloun National University, Jordan 26810, Ajloun

E-mail: g.shakah@anu.edu.jo

Abstract: Real-time computer graphics technologies such as graphics engines and graphics pipeline (software and hardware components) have improved considerably in the past decade. However, increase in efficiency and broadening of the applicability area has come at the cost of complexity of the tools. Therefore, development costs of advanced visualization systems can increase considerably. In this research we explore the possibility of addressing this problem by analyzing the software development methodology of graphics pipeline. We believe that the pipeline's limitation regarding geometrical primitives in its operations is one of the reasons for the development efficiency bottleneck. We propose an approach that extends graphics pipeline with additional processing stages that can operate with primitives of higher order. This new method has the potential to reduce the pipeline's application-level complexity for developers to create better software.

Keywords: Computer Graphics, Rendering, Pipeline, Graphics Engines, Software Development

Introduction

Computer graphics is a dynamically developing area in computer science despite being two to three decades old. Owing to the fact that people consume most of the information visually, virtually all areas of science, industry and entertainment are associated with computer graphics. Therefore, graphics has always been a primary focus of computer science inquiry and industrial advancements.

Steady growth of application areas and introduction of new technologies such as virtual reality (ACM SIGGRAPH on Virtual Reality, 2017; The Economist, 2014) and augmented reality (ACM SIGGRAPH on Augmented Reality, 2015) have increased the number of challenges in computer graphics. The efforts invested in solving these challenges are reflected in the incremental evolution of all the aspects of the central technical solution-graphics pipeline. However, as often reported in business news, the expanding complexity of the problems negatively affects the production costs of visualization software (Adward and Shreiner, 2012; Avrsr, 2018). The underlying standard technology is critically insufficient by itself for product development and many companies invest in the creation of higher level assistant software-graphics engines.

Current common approach to visualization system development often starts with problem analysis and selection of the most appropriate graphics engine suitable for the task. This is described in detail in "Visualization and Interaction in Research, Teaching and Scientific Communication" (Ammon, 2017). Given the extensive variety of engines to choose from, this may prove to be a serious problem. Additionally, in most cases, the existing software frameworks are not partially or fully applicable to the problems under consideration, which results in changes to the problems to fit the existing technologies or development of a new technological stack up from the graphics pipeline.

Throughout the years, the graphics pipeline has evolved in a process similar to precipitation: Chaotic in nature and the number of higher-level software implementations used to try out and choose the best practices, which would then be considered for standardization and incorporation into the pipeline (Tor *et al*., 2018; Shakah, 2018; Natalya, 2017). This process is ongoing and in this article, we will review the recent changes introduced into the pipeline and offer another perspective on possible steps for pipeline improvement.

Analysis of Problem

Graphics pipeline is one the central notions in computer graphics. It can be best described as a technology comprising certain hardware and software components that implement the process of rendering. Rendering is a process of data visualization using computer technology. Common rendering algorithms comprise several data transformation stages that start from an arbitrary data representation and finish with rasterized images on a display screen. Many of these stages have been standardized in the course of several years and have finally been implemented as specialized hardware components such as graphics cards.

However, specific visualization problems require a custom approach at various stages, especially at the highest application level. A special class of software called graphics engines emerges at this level. These engines can be categorized as middleware and can help software developers in their use of computer graphics. Owing to a large number of application problems, the number of available graphics engines is also considerably high because they are often attuned to specifics of the tasks they help to solve. A graphics developer must choose an appropriate graphics engine to solve a particular visualization problem. However, this approach bears the risk of distortion of the original problem because it may require adaptation to existing technology. Otherwise, if adaptation is not possible, a new specialized visualization system should be developed, which is a costly task.

Figure 1 illustrates the relative amount of implementations at each visualization stage. Standardized components of the visualization process are considerably less compared to those that are closer to end products. The number of practical problems is large; therefore, the number of products developed to address them is also large. Throughout the years, certain commonalities of visualization problems have been identified and slowly migrated towards standard components; they were initially introduced in the engines and later became a part of the pipeline's architecture.

The process of pipeline enrichment is ongoing and in this article, we will observe the newly introduced pipeline concepts that appear in DirectX12 and Vulcan graphics APIs. Additionally, we will trace a trajectory for further evolution of the higher abstraction level of graphics pipeline.

Graphics Pipeline Development

Graphics pipeline has evolved considerably. Two decades ago, it started with a couple of fixed functions programmed directly in the hardware and interfaces to tune them. The main graphics primitives of the pipeline were vertex and index data buffers that described surface geometry and topology. Early functionality of the pipeline included fixed functions for lighting calculation, which are now obsolete. It was evident that this early functionality was quite rigid and limiting for use in potential applications.

The first major step was taken for making the pipeline programmable. The goal was to supersede the fixed functions technology with a capability of changing the behavior of the pipeline stages using custom programs for particular visualization intents. These custom programs were named shaders. Initially, there were two types of shaders: vertex and pixel that were responsible for vertex and pixel (fragment) processing, respectively.

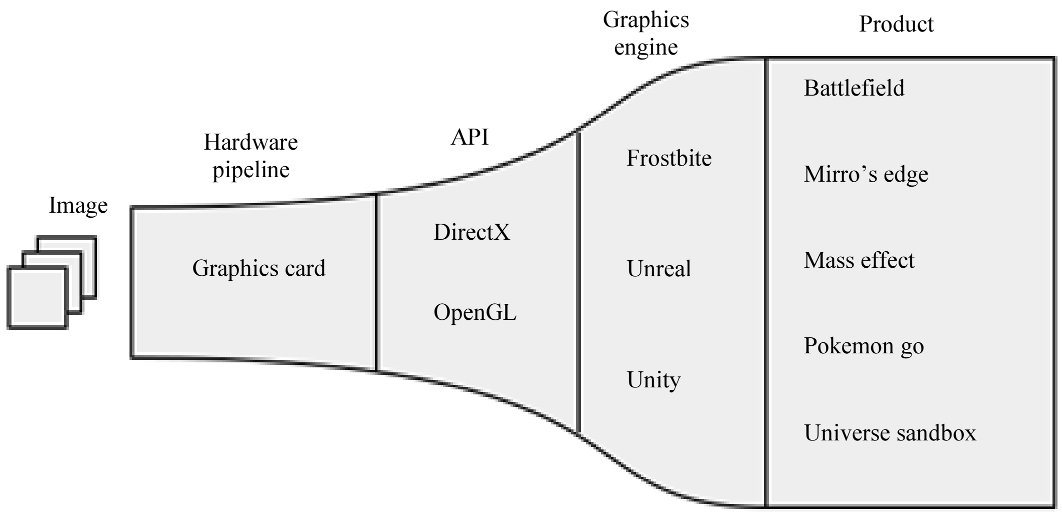

Fig. 1: Stages of visualization

The evolution of programmable graphics pipeline continued with the introduction of additional pipeline stages, additional types of shaders and extension of shaders' capabilities. Currently, the DirectX11 pipeline is defined by nine stages of which, many are programmable and some tunable (essentially still fixed-function, but no to detriment). Access to the pipeline is provided via Hardware Abstraction Layer (HAL) libraries and drivers. This layer is represented by graphics APIs of which, the most prominent two are DirectX and OpenGL. Both these libraries have followed the evolution of the pipeline since its inception and both have similar software abstractions and structure. The main difference between them is the programming paradigms used: DirectX is object-oriented, whereas OpenGL is a procedural state machine. Despite the difference, they both are used in a very similar fashion.

Recently, graphics hardware technology reached newer heights in terms of performance and the ability to process more data. These changes led to a greater refinement of the APIs, thereby uncovering flaws in the graphics hardware design and further creating a bottleneck that prevents further improvement. The main reason for the bottleneck was the ever-increasing demand for parallel computing. This new requirement was not considered in the beginning of the development of the APIs. Both the standard libraries had intrinsic limitations because both used to work in the immediate device context. In other words, the HAL instructions would be directly sent to the hardware no sooner they were called and in the same calling order. Meanwhile, a logical device model was built around a global state that was modified by instruction calls. This was one of the major roadblocks en route parallelization.

Therefore, the Microsoft and Khronos groups have introduced substantially revised versions of their APIs: DirectX12 (John and Satran, 2018; KGI, 2019) and Vulkan (Khornos, 2019; Eric *et al*., 2016) (successor of OpenGL). Newer versions of these libraries offer a new class of programming primitives that enable efficient data management and instruction flows on the pipeline. Earlier, this functionality was a part of the internal implementation of graphical drivers; however, new visualization problems demonstrated that the entire mechanics needed to be revisited (Firaxis, 2011; John, 2016). Stateless rendering is the central idea of the new approach. Its crux lies in task decomposition: one single problem can be split into a number of smaller ones that can be solved independently and later reassembled into a final solution. The stateful model did not inherently support decomposition because the state of the rendering pipeline was global and any changes made to it at earlier stages would have to be undone to revert it back to the original.

One of the most important features of the stateless model is its suitability for multithreaded execution. However, it introduces a number of new concepts that creates a requirement on the developers for advanced and overarching understanding of the graphics pipeline and related functional particularities of the hardware, right from the beginning. At the same time, the API provides much broader access to the hardware it abstracts, which enables development of applications with improved performance. The new concepts include execution queues, command buffers and various synchronization objects. The additional capabilities enable flexible work balancing on both the CPU and GPU sides. The new API is a considerable change in how the rendering process is organized on the graphics pipeline. The introduced execution granularity and state control provide new opportunities in visual systems development.

High-Level Primitives in Visualization Systems

The introduction of DirectX12 and Vulcan clearly demonstrates that the graphics pipeline development has not stopped and there is scope for intensive improvements. However, as aforementioned, the API became more complex for understanding and use, which is a fair trade-off for the flexibility it provides. Consequently, efficient usage of graphics pipeline is more challenging for new and experienced graphics developers. A common solution to this problem is graphics engines middleware.

Graphics engines usually address the problem of pipeline complexity by encapsulating it into a substitutive framework of objects and processes. The user of such visualization system is not required to know the implementation details of the engine and how it interacts with the pipeline-the engine claims to efficiently translate the user's intent into pipeline actions behind the scenes. The degree to which the graphics engine abstracts away from the pipeline's functionality differs according to the engine. However, there is a common abstraction called scene graph.

Scene graph is a common name of the data structures that are used by most graphics engines for defining the input data model ready for visualization. Scene graph has objects as nodes and relations between them as edges. Objects can define entities in a visual scene or various sets of properties such as materials. Relations can define parent-child relationships in logical or geometrical sense. The goal of the user is to represent the visualization problem in terms of a specific implementation of the scene graph. The engine's job is to translate the graph into rendering commands for the graphics pipeline. In more advanced engines, the user gets more opportunities to define different aspects of how the translation is performed. However, analogous to earlier pipeline implementations, the functionality of engines is usually at the level of the fixed-function pipeline. This implies the existence of considerable limitations to applicability and flexibility of the current visualization systems.

In our previous works we explored the possibility of introducing a different level of logical primitives (Shakah, 2019) and provided details on a possible technical implementation of the methodology using newly introduced object shaders (Krasnoproshin and Mazouka, 2014; Victor and Mazouka, 2017). Additionally, the new developments in graphics pipeline architecture strongly suggest that this approach may be the point where the current industry trends converge. Previously, we argued that graphics pipeline is functionally limited at the abstraction level of data primitives with which it operates; this means that a geometrical surface is the single largest object that the pipeline considers. Moreover, all visualization systems employ the notion of a logical object that can be represented with a collection of resources that define its look. In "An Incremental Rendering VM" (Haaser, 2015) define this representation as a simple abstraction. The goal of graphics engines in this sense is to lower the abstraction of programmer's rendering intent from the highest (product specific) to the lowest (hardware/pipeline specific) level.

Disregarding the implementation details, graphics engines translate the initial scene graph representation into lists of simple abstraction objects in a readily available format for graphics pipeline consumption.

Our approach aims at formalization of the lower level of objects' abstraction in a manner that is compatible with the rest of the graphics pipeline architecture. The model that we construct can be formalized in algebraic terms with a notion of generic objects, frames and operations that perform the transformations:

- Sample-An operation that selects a subset of objects using predefined criteria. This procedure performs task decomposition by extracting objects with particular properties that require distinct processing. For example, we can separate transparent and opaque objects by using corresponding sampling operations
- Render-An operation that transforms a set of objects and/or frames into one frame, which is essentially a subdivision of the entire visualization process. The procedure solves a visualization subtask. In a degenerate case, the entire scene can be set as input to the rendering procedure
- Blend-An operation that combines multiple frames. This procedure performs task results composition and ultimately combines the rendering outputs into the final frame

At this level of abstraction, the entire visualization problem for any system can be formulated as follows.

Given a data set of objects, build a visualization expression using algebraic operations that transform the initial data set into a single frame. This approach brings structure into the visualization process development and helps identify meaningful points of data interactions. The entire process can be represented with a data flow graph Fig. 2 that ends with a single node that plays the role of a resulting frame.

This toolset enables process and data-flow abstraction level thinking and aids development of the visualization systems. To build a visualization system for a concrete visualization problem, one needs to determine distinct classes of simple abstraction objects and their characteristics and define corresponding rendering procedures for the processes' subroutines. The procedures can then be implemented independently and later included into the complete scheme. In the event of requirements refinement, corresponding modifications can be done more efficiently in specific procedures or in the visualization process graph.

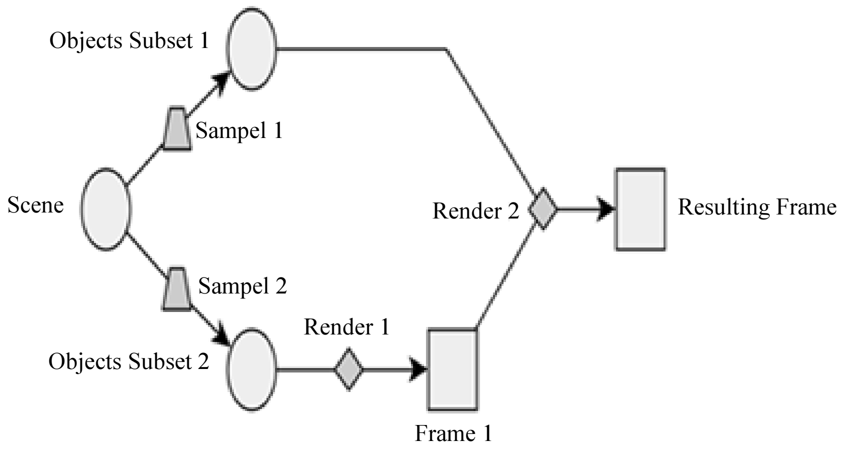

Fig. 2: Data flow graph

Generally, a higher abstraction level allows more expressive description of the developer's intent. The problem of decomposition frequently appears in visualization and currently there are no good tools in the pipeline for convenient rendering specialization. This is the main reason why developers have to implement the corresponding functionality again or use graphics engines. This results in higher development costs or a risk of the original problem's maladaptation to the engine's data model. Our approach preserves pipeline flexibility while expanding its functionality to lower usage complexity.

Decomposition also plays a substantial role in the recent DirectX12 and Vulkan developments. New primitives and procedures were added to aid stateless multithreading development in the graphics pipeline.

Development Methodology

The general methodology for applying the proposed approach comprises the following steps:

1. The initial visualization problem should be analyzed in terms of identification of smaller independent sub-problems, e.g., lighting, material shading and separation of animations into individual processes
2. Each of the smaller problems should be provided with input and output interface definition, e.g., lighting process requires information of light sources, solid objects and their reflective properties. As an output, the lighting process may generate a frame containing scene illumination
3. Each of the processes should implement a set of sample, render and blend procedures that produce the expected results, e.g., lighting process requires at least two sample procedures to extract the light sources and solid objects from the scene. The render procedure may be implemented to generate the illumination frame of a single light/object pair. The blend procedure would be required to properly use illumination in the context of the entire visualization
4. After all the sub-processes have been implemented, they should be combined as a single visualization process for the target problem

This method allows efficient deconstruction of visualization problems into smaller sub-processes that can be implemented separately and reused in later applications. Our approach offers a different perspective at visualization problems that can be completely divorced from any particular visualization system or application. Potentially, it could result in the creation of a standard library of visualization algorithms that could be used in new visualization systems at minimal developmental costs.

Recent works explicitly acknowledge the problem of development complexity but offer different solutions. In "Shader Components: Modular and High Performance Shader Development" (Yong *et al*., 2017; lake *et al*., 2000), described a method of organization of shader programs into higher units of modularity-shader components. In "Slang: language mechanisms for extensible real-time shading systems" (Yong *et al*., 2018; Thomas, 1996). built on top of their previous developments and introduce a higher-level shader language that allows for host/shader objects data reflection. The difference with our approach is that our method concentrates on the building of processes and follows the data-driven paradigm of graphics pipeline.

Conclusion

Graphics pipeline has evolved considerably by adapting to immediate problems faced by visualization systems developers. However, it is evident that the pipeline may require more significant conceptual changes by including higher level abstractions. This is because the pipeline's technical complexity expands rapidly, whereas its operational primitives remain very basic. This puts considerable pressure on product or engine developers and may lead to less stable and higher cost systems.

In the past couple of years, the computer graphics community has introduced few novel concepts into graphics pipeline that are reflected in DirectX12 and Vulcan APIs. The new capabilities provide greater control over the pipeline's process construction in multithreaded environment, which, if used correctly, will help achieve better performance of real-time rendering systems. However, this does not solve bigger issues in graphics development-general software complexity and redundancy.

Our approach to solving this problem, briefly described in this article and in a number of previous works, sets another strategic goal for pipeline evolution in a different direction. We propose a set of basic operations and primitives that encapsulate common use cases of the visualization process and are open to further automation. We believe this will help improve graphics pipeline's comprehensibility and visualization systems' maintainability while maintaining their flexibility and empowering them.

Acknowledgment

The authors would like to thank Ajloun National University for the facilities used in this work.

Author's Contributions

All the authors contributed to the final version of the manuscript.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and there are no ethical issues involved

References

ACM SIGGRAPH on Augmented Reality, 2015. SIGGRAPH on augmented reality. ACM.

ACM SIGGRAPH on Virtual Reality, 2017. SIGGRAPH on virtual reality. ACM.

Adward, A. and D. Shreiner, 2012. Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL. 6th Edn., Addison-Wesley, Boston, ISBN-10: 0132545233, pp: 730.

Ammon, C.J., 2017. Visualization and Interaction in Research, Teaching and Scientific Communication. Fall Meeting, American Geophysical Union.

AVRSR, 2018. Augmented and Virtual Reality Survey Report. AVRSR.

Eric, M., U. Hideaki and S. Fabien, 2016. Pose estimation for augmented reality: A hands-on survey. IEEE Trans.. Visualizat. Comput. Graphics 22: 12-12.

Firaxis, L., 2011. Uses of D3D11. Proceedings of the Game Developers Conference, (GDC' 11). https://www.slideserve.com/admon/firaxis-lore

Haaser, G., 2015. An incremental rendering VM. Proceedings of the 7th Conference on High-Performance Graphics Aug. 07-09, Los Angeles, California, pp: 51-60. DOI: 10.1145/2790060.2790073

John, K. and M. Satran, 2018. Microsoft direct3d 12 graphics programming guide. Important Changes from Direct3D 11 to Direct3D 12.

John, M., 2016. High performance vulkan: Lessons learned from source 2. Proceedings of the GPU,Technology Conference (GTC' 16).

KGI, 2019. Vulkan. Khronos Group Inc. https://www.khronos.org/vulkan/

Khronos. 2019. WebGL 2.0 Specification. (2019). https://www.khronos.org/registry/ webgl/specs/latest/.

Krasnoproshin, V. and D. Mazouka, 2014. High-level rendering pipeline construction tools. Proceedings of International Conference on Pattern Recognition and Information Processing (PRIP' 14).

Lake, A., C. Marshall, M. Harris and M. Blackstein, 2000. Stylized rendering techniques for scalable real-time 3D animation. Proceedings of the Symposium on Non-Photorealistic Animation and Rendering , Jun. 7-8, Annecy, France, pp: 13-20. Doi: 10.1145/340916.340918

Natalya, T., 2017. Chris tchou destiny shader pipeline. Unity Technologies, Unity 5.6 Users Manual.

The Economist, 2014. Why video games are so expensive to develop. The Economist.

Shakah, G., 2019. A new method for solving hard diagnosis problems. Comput. Eng. Intelli. Syst., 1: 13-20. DOI: 10.14419/ijet.v7i4.28039

Shakah, G., 2018. The problem of image segmentation and de-noising methods and various approaches to its solution. Int. J. Eng. Technol., 4: 5297-5301.

Thomas, W.C., 1996. Beyond the renderer: Software Architecture for Parallel Graphics and Visualization. NASA Langley Res. Center.

Tor, M.A., W.W. Fung, T.G. Rogers, 2018. General-purpose graphics processor architectures. Synthesis Lectures on Computer Architecture, 2: 1-140.

Victor, K. and D., Mazouka, 2017. Frame Manipulation Techniques in Object-Based Rendering. In: Pattern Recognition and Information Processing, Krasnoproshin, V. and S. Ablameyko (Eds.), Springer, Cham, ISBN-10: 978-3-319-54220-1, pp: 97-105.

Yong, H., K. Fatahalian and T. Foley, 2018. Slang: Language mechanisms for extensible real-time shading systems. ACM Trans. Graphics.

Yong, H., T. Foley, T. Hofstee, H. Long and K. Fatahalian, 2017. Shader components: Modular and high performance shader development. ACM Trans. Graph., 36: 4-4. DOI: 10.1145/3072959.3073648